Урок №207. Потоки ввода и вывода

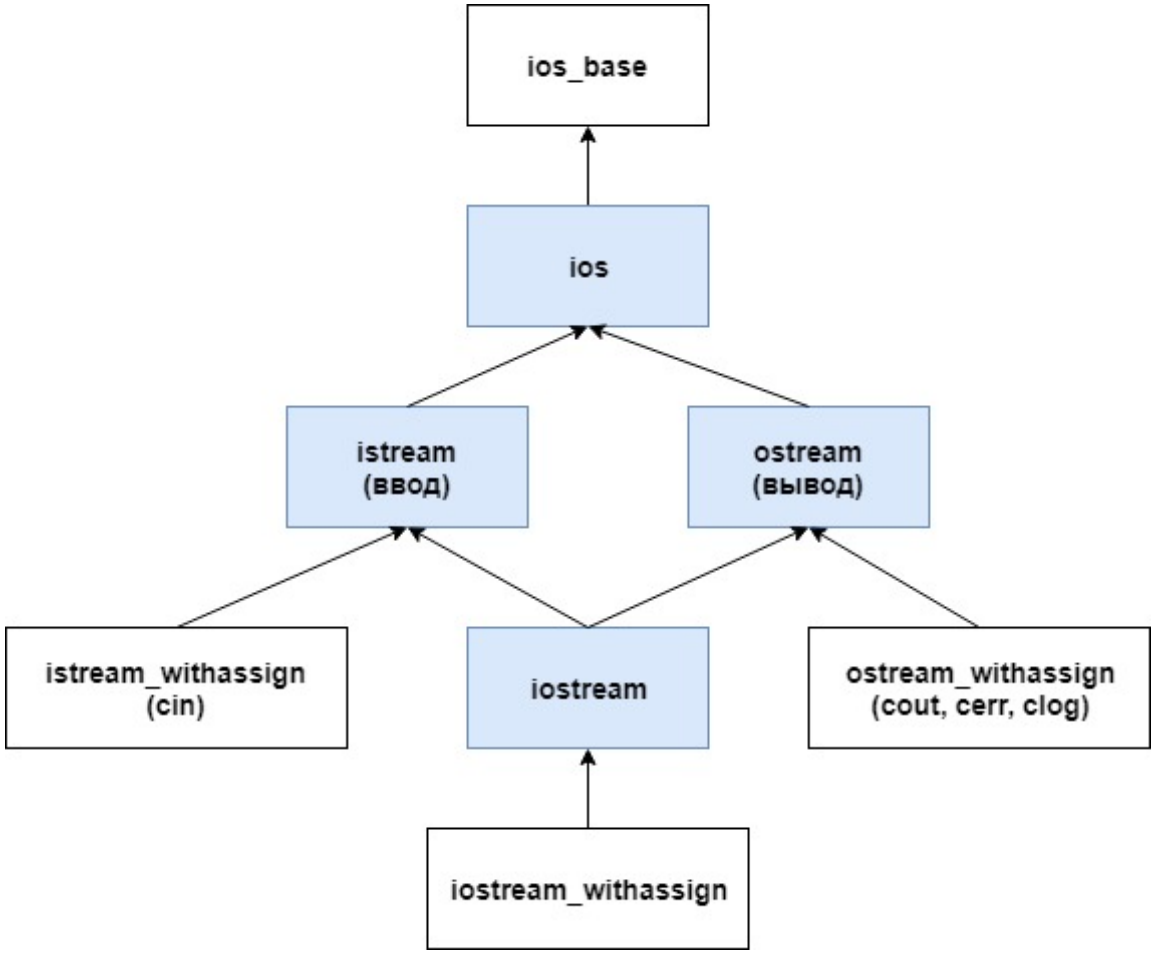
Юрий | Уроки по C++ | Обновл. 15 Сен 2021 | 57653 | 5

Функционал потоков ввода/вывода не определен как часть языка C++, а предоставляется Стандартной библиотекой C++ (и, следовательно, находится в **пространстве имен std**). На предыдущих уроках мы подключали **заголовочный файл** библиотеки iostream и использовали объекты cin и cout для простого ввода/вывода данных. На этом уроке мы детально рассмотрим библиотеку iostream.

Оглавление:

Библиотека iostream

При подключении заголовочного файла iostream, мы получаем доступ ко всей иерархии **классов** библиотеки iostream, отвечающих за функционал ввода/вывода данных (включая класс, который называется iostream). Иерархия этих классов выглядит примерно следующим образом:



Первое, что вы можете заметить в этой иерархии — **множественное наследование** (то, что на самом деле не рекомендуется использовать). Тем не менее, библиотека iostream была разработана и тщательно протестирована соответствующим образом, дабы избежать типичных ошибок, которые возникают при работе с множественным наследованием, поэтому вы можете спокойно использовать эту библиотеку.

Потоки в C++

Второе, что вы могли бы заметить — это частое использование слова *«stream»* (т.е. *«поток»*). По сути, ввод/вывод в языке C++ реализован с помощью потоков. Абстрактно, **поток** — это последовательность символов, к которым можно получить доступ. Со временем поток может производить или потреблять потенциально неограниченные объемы данных.

Мы будем иметь дело с двумя типами потоков. **Поток ввода** (или ***«входной поток»***) 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.

И наоборот, **поток вывода** (или ***«выходной поток»***) 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.

Хорошая новость заключается в том, что программисту не нужно знать детали взаимодействия потоков с разными устройствами и источниками данных, ему нужно только научиться взаимодействовать с этими потоками для чтения и записи данных.

Ввод/вывод в C++

Хотя класс `ios` является **дочерним** классу `ios_base`, очень часто именно этот класс будет наиболее родительским классом, с которым вы будете работать/взаимодействовать напрямую. Класс `ios` определяет кучу разных вещей, которые являются общими для 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. Это также имеет смысл: вы вставляете свои значения в поток, а затем потребитель данных (например, монитор) использует их.

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

Наконец, остались 3 класса, оканчивающиеся на `_withassign`. Эти потоковые классы являются дочерними классам `istream`, `ostream` и `iostream` (соответственно). В большинстве случаев вы не будете работать с ними напрямую.

Стандартные потоки в C++

Стандартный поток — это предварительно подключенный поток, который предоставляется программе её окружением. Язык C++ поставляется с 4-мя предварительно определенными стандартными объектами потоков, которые вы можете использовать (первые три вы уже встречали):

- **cin** — класс `istream_withassign`, связанный со стандартным вводом (обычно это клавиатура);
- **cout** — класс `ostream_withassign`, связанный со стандартным выводом (обычно это монитор);
- **cerr** — класс `ostream_withassign`, связанный со стандартной ошибкой (обычно это монитор), обеспечивающий небуферизованный вывод;
- **clog** — класс `ostream_withassign`, связанный со стандартной ошибкой (обычно это монитор), обеспечивающий буферизованный вывод.

Небуферизованный вывод обычно обрабатывается сразу же, тогда как буферизованный вывод обычно сохраняется и выводится как блок. Поскольку `clog` используется редко, то его обычно игнорируют.

Пример на практике

Вот пример использования ввода/вывода данных со стандартными потоками:

```cpp
#include <iostream>
#include <cstdlib> // для exit()

int main()
{
    // Сначала мы используем оператор вставки с объектом cout для вывода текста на монитор
    std::cout << "Enter your age: " << std::endl;

    // Затем мы используем оператор извлечения с объектом cin для получения пользовательского ввода
    int nAge;
    std::cin >> nAge;

    if (nAge <= 0)
    {
        // В этом случае мы используем оператор вставки с объектом cerr для вывода сообщения об ошибке
        std::cerr << "Oops, you entered an invalid age!" << std::endl;
        exit(1);
    }

    // А здесь мы используем оператор вставки с объектом cout для вывода результата
    std::cout << "You entered " << nAge << " years old" << std::endl;

    return 0;
}
```

На следующих уроках мы детально рассмотрим функционал потоков ввода/вывода.

Оценить статью:

(**126** оценок, среднее: **4,80** из 5)

Поделиться в социальных сетях:

Урок №206. Вставка символов и строк в std::string

Урок №208. Функционал класса istream

Комментариев: 5

VALET_PIKOVIY

14 ноября 2021 в 20:34

Создатели этого сайта, хочу вам сказать что вы очень крутые ребята , вы сделали очень полезные уроки по программированию, отдельное спасибо за практику) От души:)

Урок №208. Функционал класса istream

Юрий | Уроки по C++ | Обновл. 15 Сен 2021 | 34433 | 2

Библиотека iostream по своей сути довольно сложная, поэтому мы не сможем охватить её полностью в рамках данных уроков. Тем не менее, мы можем рассмотреть её основной функционал. На этом уроке мы разберемся с классом `istream`.

> ***Примечание***: Весь функционал объектов, которые работают с **потоками ввода/вывода**, находится в **пространстве имен std**. Это означает, что вам нужно либо добавлять префикс `std::` ко всем объектам и функциям ввода/вывода, либо использовать строку `using namespace std;`.

Оглавление:

Оператор извлечения

Как вы уже узнали на предыдущем уроке, мы можем использовать оператор извлечения `>>` для считывания информации из входного потока. Одной из наиболее распространенных проблем при считывании строк из входного потока является предотвращение **переполнения**. Например:

```
#include <iostream>

int main()
{
    char buf[12];
    std::cin >> buf;
}
```

Что произойдет, если пользователь введет 20 символов? Правильно, 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 (`<<`).

Мы уже работали с одним из манипуляторов — endl, который одновременно выводит символ новой строки и удаляет текущие данные из буфера. Язык C++ предоставляет еще один манипулятор — **setw()** (из заголовочного файла iomanip), который используется для ограничения количества символов, считываемых из потока. Для использования setw() вам нужно просто передать в качестве параметра максимальное количество символов для извлечения и вставить вызов этого манипулятора следующим образом:

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

int main()
{
    char buf[12];
    std::cin >> std::setw(12) >> buf;
}
```

Эта программа теперь прочитает только первые 11 символов из входного потока (+ один символ для **нуль-терминатора**). Все остальные символы останутся в потоке до следующего извлечения.

Извлечение и пробелы

Важный момент: оператор извлечения работает с «отформатированными» данными, т.е. он игнорирует все пробелы, символы табуляции и символ новой строки. Например:

```
#include <iostream>

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

```

```
    return 0;
}
```

Если пользователь введет следующее:

```
Hello! My name is Anton
```

То оператор извлечения пропустит все пробелы и символы новой строки. Следовательно, результат выполнения программы:

```
Hello!MynameisAnton
```

Часто пользовательский ввод все же нужен со всеми его пробелами. Для этого класс `istream` предоставляет множество функций. Одной из наиболее полезных является **фунция get()**, которая извлекает символ из входного потока. Вот вышеприведенная программа, но уже с использованием функции get():

```
#include <iostream>

int main()
{
    char ch;
    while (std::cin.get(ch))
        std::cout << ch;

    return 0;
}
```

Теперь, если мы введем следующее:

```
Hello! My name is Anton
```

То получим:

```
Hello! My name is Anton
```

Функция get() также имеет строковую версию, в которой можно указать максимальное количество символов для извлечения. Например:

```
#include <iostream>

int main()
{
    char strBuf[12];
    std::cin.get(strBuf, 12);
    std::cout << strBuf << std::endl;

    return 0;
}
```

Если мы введем следующее:

```
Hello! My name is Anton
```

То получим:

```
Hello! My n
```

Обратите внимание, программа считывает только первые 11 символов (+ нуль-терминатор). Остальные символы остаются во входном потоке.

Один важный нюанс: функция get() не считывает символ новой строки! Например:

```
#include <iostream>

int main()
{
```

```
    char strBuf[12];

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

    // Считываем дополнительно еще 11 символов
    std::cin.get(strBuf, 12);
    std::cout << strBuf << std::endl;
    return 0;
}
```

Если пользователь введет следующее:

`Hello!`

То получит:

`Hello!`

И программа сразу же завершит свое выполнение! Почему так? Почему не срабатывает второй ввод данных? Дело в том, что первый get() считывает символы до символа новой строки, а затем останавливается. Второй get() видит, что во входном потоке все еще есть данные и пытается их извлечь. Но первый символ, на который он натыкается — символ новой строки, поэтому происходит второй «Стоп!».

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:

```
#include <iostream>

int main()
{
    char strBuf[12];
```

```
6
7       // Считываем 11 символов
8       std::cin.getline(strBuf, 12);
9       std::cout << strBuf << std::endl;
10
11      // Считываем дополнительно еще 11 символов
12      std::cin.getline(strBuf, 12);
13      std::cout << strBuf << std::endl;
14      return 0;
15  }
```

Этот код работает точно так, как ожидается, даже если пользователь введет строку с символом новой строки.

Если вам нужно узнать количество символов, извлеченных последним getline(), используйте **функцию gcount()**:

```
1   #include <iostream>
2
3   int main()
4   {
5       char strBuf[100];
6       std::cin.getline(strBuf, 100);
7       std::cout << strBuf << std::endl;
8       std::cout << std::cin.gcount() << " characters were read" << std::endl;
9
10      return 0;
11  }
```

Результат:

```
Hello! My name is Anton
24 characters were read
```

Специальная версия функции getline() для std::string

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. Например:

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

int main()
{
    using namespace std;

    string strBuf;
    getline(cin, strBuf);
    cout << strBuf << endl;

    return 0;
}
```

Еще несколько полезных функций класса istream

Есть еще несколько полезных функций класса `istream`, которые вы можете использовать:

- **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 вами символ обратно в поток, чтобы его можно было извлечь в следующий раз.

Класс `istream` содержит еще множество других полезный функций и их вариаций, но это уже тема для отдельного туториала.

Оценить статью:

 (**119** оценок, среднее: **4,87** из 5)

Поделиться в социальных сетях:

Урок №207. Потоки ввода и вывода

Урок №209. Функционал классов ostream и ios. Форматирование вывода

Урок №209. Функционал классов ostream и ios. Форматирование вывода

Юрий | Уроки по C++ | Обновл. 15 Сен 2021 | 33232 | 4

На этом уроке мы рассмотрим функционал классов `ostream` и `ios` в языке C++.

> ***Примечание***: Весь функционал объектов, которые работают с **потоками ввода/вывода**, находится в **пространстве имен std**. Это означает, что вам нужно либо добавлять префикс `std::` ко всем объектам и функциям ввода/вывода, либо использовать в программе строку `using namespace std;`.

Оглавление:

Форматирование вывода

Оператор вставки (вывода) `<<` используется для помещения информации в выходной поток. А как мы уже знаем из урока о потоках, классы `istream` и `ostream` являются **дочерними** классу `ios`. 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*;
- **манипуляторы** — это объекты, которые помещаются в поток и влияют на способ ввода/вывода данных.

Для включения флага используйте **функцию setf()** с соответствующим флагом в качестве параметра. Например, по умолчанию C++ не выводит знак `+` перед положительными числами. Однако, используя флаг std::showpos, мы можем это изменить:

```
1 #include <iostream>
```

```
2
3 int main()
4 {
5     std::cout.setf(std::ios::showpos); // включаем флаг std::showpos
6     std::cout << 30 << '\n';
7 }
```

Результат:

```
+30
```

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

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout.setf(std::ios::showpos | std::ios::uppercase); // включаем флаги std::showpos и std::uppercase
6     std::cout << 30 << '\n';
7 }
```

Чтобы отключить флаг, используйте функцию **unsetf()**:

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout.setf(std::ios::showpos); // включаем флаг std::showpos
6     std::cout << 30 << '\n';
7     std::cout.unsetf(std::ios::showpos); // выключаем флаг std::showpos
8     std::cout << 31 << '\n';
9 }
```

Результат:

```
+30
31
```

Многие флаги принадлежат к определенным группам форматирования. **Группа форматирования** — это группа флагов, которые задают аналогичные (иногда взаимоисключающие) параметры форматирования вывода. Например, есть группа форматирования basefield.

Флаги группы форматирования basefield:

- **oct** (от англ. «***oct****al*» = «*восьмеричный*») — восьмеричная система счисления;
- **dec** (от англ. «***dec****imal*» = «*десятичный*») — десятичная система счисления;
- **hex** (от англ. «***hex****adecimal*» = «*шестнадцатеричный*») — шестнадцатеричная система счисления.

Эти флаги управляют выводом целочисленных значений. По умолчанию установлен флаг std::dec, т.е. значения выводятся в десятичной системе счисления. Попробуем сделать следующее:

```
#include <iostream>

int main()
{
    std::cout.setf(std::ios::hex); // включаем флаг std::hex
    std::cout << 30 << '\n';
}
```

Результат:

```
30
```

Ничего не работает! Почему? Дело в том, что setf() только включает флаги, он не настолько умен, чтобы одновременно отключать другие (взаимоисключающие) флаги. Следовательно, когда мы включаем std::hex, std::dec

также включен и у него приоритет выше. Есть два способа решения данной проблемы.

Во-первых, мы можем отключить std::dec, а затем включить std::hex:

```
#include <iostream>

int main()
{
    std::cout.unsetf(std::ios::dec); // выключаем вывод в десятичной системе счисления
    std::cout.setf(std::ios::hex); // включаем вывод в шестнадцатеричной системе счисления
    std::cout << 30 << '\n';
}
```

Теперь уже результат тот, что нужно:

```
1e
```

Второй способ — использовать вариацию функции setf(), которая принимает два параметра:

- первый параметр — это флаг, который нужно включить/выключить;
- второй параметр — группа форматирования, к которой принадлежит флаг.

При использовании этой вариации функции setf() все флаги, принадлежащие группе форматирования, отключаются, а включается только передаваемый флаг. Например:

```
#include <iostream>

int main()
{
    // Включаем и оставляем включенным единственный флаг (std::hex) группы форматирования std::basefield
    std::cout.setf(std::ios::hex, std::ios::basefield);
    std::cout << 30 << '\n';
}
```

Результат:

```
1e
```

Язык C++ также предоставляет еще один способ изменения параметров форматирования: манипуляторы. Фишка манипуляторов в том, что они достаточно умны, чтобы одновременно включать и выключать соответствующие флаги. Например:

```
#include <iostream>

int main()
{
    std::cout << std::hex << 30 << '\n'; // выводим 30 в шестнадцатеричной системе счисления
    std::cout << 31 << '\n'; // мы все еще находимся в шестнадцатеричной системе счисления
    std::cout << std::dec << 32 << '\n'; // перемещаемся обратно в десятичную систему счисления
}
```

Результат:

```
1e
1f
32
```

В общем, использовать манипуляторы гораздо проще, нежели включать/выключать флаги. Многие параметры форматирования можно изменять как через флаги, так и через манипуляторы, но есть и такие параметры форматирования, которые изменить можно либо только через флаги, либо только через манипуляторы.

Полезные флаги, манипуляторы и методы

Ниже мы рассмотрим список наиболее полезных флагов, манипуляторов и методов. Флаги находятся в классе `ios`, манипуляторы — в пространстве имен std, а методы — в классе `ostream`.

Флаг:

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

Манипуляторы:

- **boolalpha** — логические значения выводятся как `true` / `false`.
- **noboolalpha** — логические значения выводятся как `0` / `1`.

Например:

```
#include <iostream>

int main()
{
    std::cout << true << " " << false << '\n';

    std::cout.setf(std::ios::boolalpha);
    std::cout << true << " " << false << '\n';

    std::cout << std::noboolalpha << true << " " << false << '\n';

    std::cout << std::boolalpha << true << " " << false << '\n';
}
```

Результат:

```
1 0
true false
1 0
true false
```

Флаг:

- **showpos** — если включен, то перед положительными числами указывается знак `+`.

Манипуляторы:

- **showpos** — перед положительными числами указывается знак `+`.
- **noshowpos** — перед положительными числами не указывается знак `+`.

Например:

```cpp
#include <iostream>

int main()
{
    std::cout << 7 << '\n';

    std::cout.setf(std::ios::showpos);
    std::cout << 7 << '\n';

    std::cout << std::noshowpos << 7 << '\n';

    std::cout << std::showpos << 7 << '\n';
}
```

Результат:

```
7
+7
7
+7
```

Флаг:

- **uppercase** — если включен, то используются заглавные буквы.

Манипуляторы:

- **uppercase** — используются заглавные буквы.
- **nouppercase** — используются строчные буквы.

Например:

```cpp
#include <iostream>

int main()
{
    std::cout << 12345678.9 << '\n';

    std::cout.setf(std::ios::uppercase);
    std::cout << 12345678.9 << '\n';

    std::cout << std::nouppercase << 12345678.9 << '\n';

    std::cout << std::uppercase << 12345678.9 << '\n';
}
```

Результат:

```
1.23457e+007
1.23457E+007
1.23457e+007
1.23457E+007
```

Флаги группы форматирования basefield:

- **dec** — значения выводятся в десятичной системе счисления;
- **hex** — значения выводятся в шестнадцатеричной системе счисления;
- **oct** — значения выводятся в восьмеричной системе счисления.

Манипуляторы:

- **dec** — значения выводятся в десятичной системе счисления;
- **hex** — значения выводятся в шестнадцатеричной системе счисления;
- **oct** — значения выводятся в восьмеричной системе счисления.

Например:

```cpp
#include <iostream>

int main()
{
    std::cout << 30 << '\n';

    std::cout.setf(std::ios::dec, std::ios::basefield);
    std::cout << 30 << '\n';

    std::cout.setf(std::ios::oct, std::ios::basefield);
```

```
    std::cout << 30 << '\n';

    std::cout.setf(std::ios::hex, std::ios::basefield);
    std::cout << 30 << '\n';

    std::cout << std::dec << 30 << '\n';
    std::cout << std::oct << 30 << '\n';
    std::cout << std::hex << 30 << '\n';
}
```

Результат:

```
30
30
36
1e
30
36
1e
```

Теперь вы уже должны понимать связь между флагами и манипуляторами.

Точность, запись чисел и десятичная точка

Используя манипуляторы (или флаги), можно изменить точность и формат вывода значений **типа с плавающей точкой**.

Флаги группы форматирования floatfield:

- **fixed** — используется десятичная запись чисел типа с плавающей точкой;

- **scientific** — используется **экспоненциальная запись** чисел типа с плавающей точкой;
- **showpoint** — всегда отображается десятичная точка и конечные нули для чисел типа с плавающей точкой.

Манипуляторы:

- **fixed** — используется десятичная запись значений;
- **scientific** — используется экспоненциальная запись значений;
- **showpoint** — отображается десятичная точка и конечные нули чисел типа с плавающей точкой;
- **noshowpoint** — не отображаются десятичная точка и конечные нули чисел типа с плавающей точкой;
- **setprecision(int)** — задаем точность для чисел типа с плавающей точкой.

Методы:

- **precision()** — возвращаем текущую точность для чисел типа с плавающей точкой;
- **precision(int)** — задаем точность для чисел типа с плавающей точкой.

Если используется десятичная или экспоненциальная запись чисел, то точность определяет количество цифр после запятой/точки. Обратите внимание, если точность меньше количества значащих цифр, то число будет округлено. Например:

```
#include <iostream>
#include <iomanip> // для setprecision()

int main()
{
    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';
}
```

Результат:

```
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. Например:

```
#include <iostream>
#include <iomanip> // для setprecision()
```

```
3
4  int main()
5  {
6      std::cout << std::setprecision(3) << 123.456 << '\n';
7      std::cout << std::setprecision(4) << 123.456 << '\n';
8      std::cout << std::setprecision(5) << 123.456 << '\n';
9      std::cout << std::setprecision(6) << 123.456 << '\n';
10     std::cout << std::setprecision(7) << 123.456 << '\n';
11 }
```

Результат:

```
123
123.5
123.46
123.456
123.456
```

Используя манипулятор или флаг showpoint, мы можем заставить программу выводить десятичную точку и конечные нули. Например:

```
1  #include <iostream>
2  #include <iomanip> // для setprecision()
3
4  int main()
5  {
6      std::cout << std::showpoint;
7      std::cout << std::setprecision(3) << 123.456 << '\n';
8      std::cout << std::setprecision(4) << 123.456 << '\n';
9      std::cout << std::setprecision(5) << 123.456 << '\n';
10     std::cout << std::setprecision(6) << 123.456 << '\n';
11     std::cout << std::setprecision(7) << 123.456 << '\n';
12 }
```

Результат:

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

Ширина поля, символы-заполнители и выравнивание

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).

Флаги группы форматирования adjustfield:

- **internal** — знак значения выравнивается по левому краю, а само значение — по правому краю;
- **left** — значение и его знак выравниваются по левому краю;
- **right** — значение и его знак выравниваются по правому краю.

Манипуляторы:

- **internal** — знак значения выравнивается по левому краю, а само значение — по правому краю;
- **left** — значение и его знак выравниваются по левому краю;
- **right** — значение и его знак выравниваются по правому краю;
- **setfill(char)** — задаем символ-заполнитель;

- **setw(int)** — задаем ширину поля.

Методы:

- **fill()** — возвращаем текущий символ-заполнитель;
- **fill(char)** — задаем новый символ-заполнитель;
- **width()** — возвращаем текущую ширину поля;
- **width(int)** — задаем ширину поля.

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

```
#include <iostream>
#include <iomanip> // для setw()

int main()
{
    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
}
```

Результат:

```
-12345
   -12345
-12345
```

```
4      -12345
5  -    12345
```

Теперь давайте зададим свой собственный символ-заполнитель:

```
1  #include <iostream>
2  #include <iomanip> // для setw()
3
4  int main()
5  {
6      std::cout.fill('*');
7      std::cout << -12345 << '\n'; // выводим значение без использования ширины поля
8      std::cout << std::setw(10) << -12345 << '\n'; // выводим значение с использованием ширины поля
9      std::cout << std::setw(10) << std::left << -12345 << '\n'; // выравниваем по левому краю
10     std::cout << std::setw(10) << std::right << -12345 << '\n'; // выравниваем по правому краю
11     std::cout << std::setw(10) << std::internal << -12345 << '\n'; // знак значения выравнивается по левому краю, а
12 }
```

Результат:

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

Обратите внимание, всё пустое пространство вокруг чисел заполнено `*` (символом-заполнителем).

Класс `ostream` и библиотека iostream содержат и другие полезные функции, флаги и манипуляторы. Но, как и в случае с классом `istream`, рассмотреть их все в рамках данного урока мы не можем. Однако основной функционал и общее представление вы получили.

Оценить статью:

(**110** оценок, среднее: **4,95** из 5)

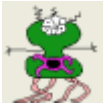
Поделиться в социальных сетях:

Vkontakte Telegram Facebook Twitter Pinterest Odnoklassniki

Урок №208. Функционал класса istream

Урок №210. Потоковые классы и Строки

Комментариев: 4



somebox

14 августа 2019 в 16:38

Урок №210. Потоковые классы и Строки

Юрий | Уроки по C++ | Обновл. 15 Сен 2021 | 39956 | 3

В Стандартной библиотеке C++ есть отдельный набор **классов**, которые позволяют использовать уже знакомые нам операторы вставки (`<<`) и извлечения (`>>`) со строками.

Оглавление:

3. **Очистка stringstream для повторного использования**

Потоковые классы

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.

Чтобы использовать **класс stringstream**, нужно подключить **заголовочный файл** sstream. Чтобы добавить данные в stringstream, мы можем использовать оператор вставки (`<<`):

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

int main()
{
```

```
    std::stringstream myString;
    myString << "Lorem ipsum!" << std::endl; // вставляем "Lorem ipsum!" в stringstream
}
```

Либо **функцию str(string)**:

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

int main()
{
    std::stringstream myString;
    myString.str("Lorem ipsum!"); // присваиваем буферу stringstream значение "Lorem ipsum!"
}
```

Аналогично, чтобы получить данные обратно из stringstream, мы можем использовать функцию str():

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

int main()
{
    std::stringstream myString;
    myString << "336000 12.14" << std::endl;
    std::cout << myString.str();
}
```

Результат:

```
336000 12.14
```

Либо оператор извлечения (`>>`):

```
#include <iostream>
```

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

int main()
{
    std::stringstream myString;
    myString << "336000 12.14"; // вставляем (числовую) строку в поток

    std::string part1;
    myString >> part1;

    std::string part2;
    myString >> part2;

    // Выводим числа
    std::cout << part1 << " and " << part2 << std::endl;
}
```

Результат:

```
336000 and 12.14
```

Обратите внимание, оператор извлечения (`>>`) перебирает буфер, автоматически разбивая его на отдельные значения с помощью имеющихся пробелов (т.е. одно использование оператора извлечения (`>>`) равно одному значению из буфера). В то время, как функция str() возвращает все данные из потока (не частично, а полностью), даже если перед ней использовался оператор извлечения.

Конвертация строк в числа и наоборот

Мы можем использовать операторы вставки и извлечения со строками для их конвертации в числа и наоборот. Например, конвертация чисел в строки:

```

```

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

int main()
{
    std::stringstream myString;

    int nValue = 336000;
    double dValue = 12.14;
    myString << nValue << " " << dValue;

    std::string strValue1, strValue2;
    myString >> strValue1 >> strValue2;

    std::cout << strValue1 << " " << strValue2 << std::endl;
}
```

Результат:

```
336000 12.14
```

А теперь конвертация (числовой) строки обратно в числа:

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

int main()
{
    std::stringstream myString;
    myString << "336000 12.14"; // вставляем (числовую) строку в поток
    int nValue;
    double dValue;

    myString >> nValue >> dValue;
```

```
12
13      std::cout << nValue << " " << dValue << std::endl;
14  }
```

Результат:

```
336000 12.14
```

Очистка stringstream для повторного использования

Есть несколько способов очистить буфер stringstream:

Способ №1: Использовать функцию str() с пустой **строкой C-style**:

```
1  #include <iostream>
2  #include <sstream> // для stringstream
3
4  int main()
5  {
6      std::stringstream myString;
7      myString << "Hello ";
8
9      myString.str(""); // очищаем буфер
10
11     myString << "World!";
12     std::cout << myString.str();
13 }
```

Способ №2: Использовать функцию str() с пустым объектом **std::string**:

```
1  #include <iostream>
2  #include <sstream> // для stringstream
3
```

```
int main()
{
    std::stringstream myString;
    myString << "Hello ";

    myString.str(std::string()); // очищаем буфер

    myString << "World!";
    std::cout << myString.str();
}
```

Результат выполнения вышеприведенных программ:

```
World!
```

При очистке stringstream неплохой идеей является вызов функции clear():

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

int main()
{
    std::stringstream myString;
    myString << "Hello ";

    myString.str(""); // очищаем буфер
    myString.clear(); // сбрасываем все флаги ошибок

    myString << "World!";
    std::cout << myString.str();
}
```

Функция clear() сбрасывает все флаги ошибок, которые были ранее установлены, и возвращает поток обратно в его прежнее (без ошибок) состояние. Мы поговорим детально о состояниях потока и флагах ошибок на следующем

уроке.

Оценить статью:

(**103** оценок, среднее: **4,92** из 5)

Поделиться в социальных сетях:

Vkontakte Telegram Facebook Twitter Pinterest Odnoklassniki

Урок №209. Функционал классов ostream и ios. Форматирование вывода

Урок №211. Состояния потока и валидация пользовательского ввода

Комментариев: 3

Урок №211. Состояния потока и валидация пользовательского ввода

Юрий | Уроки по C++ | Обновл. 15 Сен 2021 | 24365 | 6

На этом уроке мы рассмотрим состояния потока, валидацию пользовательского ввода и какой она бывает, а также нюансы, связанные с этой темой в языке C++.

Оглавление:

Состояния потока

Класс **ios_base** содержит следующие **флаги для обозначения состояния потоков**:

- **goodbit** — всё хорошо;
- **badbit** — произошла какая-то фатальная ошибка (например, программа попыталась прочитать данные после конца файла);
- **eofbit** — поток достиг конца файла;
- **failbit** — произошла какая-то НЕ фатальная ошибка (например, пользователь ввел буквы, когда программа ожидала числа).

Хотя эти флаги находятся в `ios_base`, но поскольку **ios** является **дочерним** классом для `ios_base`, доступ к этим флагам также возможен и через `ios` (например, как `std::ios::failbit`).

`ios` также предоставляет **ряд методов для доступа к вышеперечисленным состояниям потока**:

- **good()** — возвращает `true`, если установлен goodbit (значит, что с потоком всё ок);

- **bad()** — возвращает `true`, если установлен badbit (значит, что произошла какая-то фатальная ошибка);
- **eof()** — возвращает `true`, если установлен eofbit (значит, что поток находится в конце файла);
- **fail()** — возвращает `true`, если установлен failbit (значит, что произошла какая-то НЕ фатальная ошибка);
- **clear()** — сбрасывает все текущие флаги состояния потока и задает ему goodbit;
- **clear(state)** — сбрасывает все текущие флаги состояния потока и устанавливает флаг, переданный в качестве параметра (state);
- **rdstate()** — возвращает текущие установленные флаги;
- **setstate(state)** — устанавливает флаг состояния, переданный в качестве параметра (state).

Чаще всего мы будем иметь дело с failbit, который срабатывает при некорректном пользовательском вводе. Например:

```
#include <iostream>

int main()
{
    std::cout << "Enter your age: ";
    int nAge;
    std::cin >> nAge;
}
```

Обратите внимание, эта программа ожидает от пользователя ввод целого числа. Однако, если пользователь введет что-либо другое (например, `Tom`), то cin не сможет извлечь что-либо в `nAge`, и для потока будет установлен флаг failbit.

Если же возникает ошибка, и для потока задан какой-либо другой флаг (отличный от goodbit), то дальнейшие операции с этим потоком будут проигнорированы. Это можно исправить, вызвав метод clear().

Валидация пользовательского ввода

Валидация пользовательского ввода — это процесс проверки того, соответствует ли пользовательский ввод заданным критериям. Обычно, валидация ввода бывает числовой и строковой.

Со **строковой валидацией** мы принимаем весь пользовательский ввод в качестве строки, а затем либо принимаем эту строку, либо отклоняем её (в зависимости от критериев проверки). Например, если мы просим пользователя ввести номер телефона, то мы должны убедиться, что этот номер состоит из 10 цифр. В большинстве языков (особенно в скриптовых, таких как Perl и PHP) это можно сделать с помощью **регулярных выражений**. В языке C++ нет встроенной поддержки регулярных выражений (возможно, это добавят в следующих версиях языка C++), поэтому обычно это делается путем проверки каждого символа строки на соответствие заданным критериям.

С **числовой валидацией** мы обычно заботимся о том, чтобы число, которое ввел пользователь, находилось в определенном диапазоне (например, от 0 до 20). Однако, в отличие от строковой валидации, пользователь может ввести данные, которые вообще не являются числами, а нам нужно будет обрабатывать и такие случаи.

Для решения этой проблемы C++ предоставляет ряд полезных функций, которые мы можем использовать для определения того, являются ли конкретные символы цифрами или буквами. Следующие функции находятся в **заголовочном файле** cctype:

- **функция isalnum(int)** — возвращает ненулевое значение, если параметром является буква или цифра;
- **функция isalpha(int)** — возвращает ненулевое значение, если параметром является буква;
- **функция iscntrl(int)** — возвращает ненулевое значение, если параметром является управляющий символ;
- **функция isdigit(int)** — возвращает ненулевое значение, если параметром является цифра;
- **функция isgraph(int)** — возвращает ненулевое значение, если параметром является выводимый символ (но не пробел);

- **функция isprint(int)** — возвращает ненулевое значение, если параметром является выводимый символ, включая пробел;

- **функция ispunct(int)** — возвращает ненулевое значение, если параметром не являются ни буква, ни цифра, ни пробел;

- **функция isspace(int)** — возвращает ненулевое значение, если параметром является пробел;

- **функция isxdigit(int)** — возвращает ненулевое значение, если параметром является шестнадцатеричная цифра (`0-9`, `a-f`, `A-F`).

Строковая валидация

Давайте выполним простую строковую валидацию, попросив пользователя ввести свое имя. Наши ограничения: имя может содержать только буквы и пробелы. Если пользователь введет что-либо лишнее, то ввод отклоняется. Перебирать и проверять мы будем каждый символ пользовательского ввода:

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

int main()
{
    while (1)
    {
        // Просим пользователя ввести свое имя
        std::cout << "Enter your name: ";
        std::string strName;
        std::getline(std::cin, strName); // извлекаем целую строку, включая пробелы

        bool bRejected = false;

```

```
        // Перебираем каждый символ строки до тех пор, пока не дойдем до конца строки или до отклонения символа
        for (unsigned int nIndex = 0; nIndex < strName.length() && !bRejected; ++nIndex)
        {
            // Если текущий символ является буквой, то всё ок
            if (isalpha(strName[nIndex]))
                continue;

            // Если пробел, то тоже ок
            if (strName[nIndex] == ' ')
                continue;

            // В противном случае, отклоняем весь пользовательский ввод
            bRejected = true;
        }

        // Если пользовательский ввод был принят, то мы выходим из цикла while, и программа завершает свое выполнени
        // В противном случае, мы просим пользователя ввести свое имя еще раз
        if (!bRejected)
            break;
    }
}
```

Обратите внимание, этот код не идеален: пользователь может ввести в качестве своего имени `djskbvjdb jdhsbj js` или вообще одни пробелы. Мы можем усилить валидацию, уточнив наши критерии проверки: имя пользователя должно содержать как минимум 1 символ и не более одного пробела.

Теперь рассмотрим другой случай, когда мы просим пользователя ввести свой номер телефона. В отличие от имени пользователя, номер телефона имеет фиксированную длину. Следовательно, мы будем использовать другой подход к валидации пользовательского ввода. Мы напишем функцию, которая будет проверять номер телефона, который ввел пользователь, на соответствие заранее определенному шаблону (такой вот своеобразный аналог регулярным выражениям).

Шаблон будет работать следующим образом:

- `#` — любая цифра в пользовательском вводе;
- `@` — любая буква в пользовательском вводе;
- `_` — любой пробел в пользовательском вводе;
- `?` — вообще любой символ.

Все символы пользовательского ввода и нашего шаблона должны точно совпадать.

Итак, если мы хотим, чтобы пользовательский ввод соответствовал шаблону `(###) ###-####`, то пользователь должен ввести: `(` три цифры `)`, пробел, три цифры, тире и еще четыре цифры. Если что-то из этого не совпадет, то пользовательский ввод будет отклонен, например:

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

bool InputMatches(std::string strUserInput, std::string strTemplate)
{
    if (strTemplate.length() != strUserInput.length())
        return false;

    // Перебираем каждый символ пользовательского ввода
    for (unsigned int nIndex = 0; nIndex < strTemplate.length(); nIndex++)
    {
        switch (strTemplate[nIndex])
        {
        case '#': // = цифра
            if (!isdigit(strUserInput[nIndex]))
                return false;
            break;
```

```
        case '_': // = пробел
            if (!isspace(strUserInput[nIndex]))
                return false;
            break;
        case '@': // = буква
            if (!isalpha(strUserInput[nIndex]))
                return false;
            break;
        case '?': // = вообще любой символ
            break;
        default: // = точное совпадение с символом
            if (strUserInput[nIndex] != strTemplate[nIndex])
                return false;
        }
    }

    return true;
}

int main()
{
    std::string strValue;

    while (1)
    {
        std::cout << "Enter a phone number (###) ###-####: ";
        std::getline(std::cin, strValue); // извлекаем целую строку, включая пробелы
        if (InputMatches(strValue, "(###) ###-####"))
            break;
    }

    std::cout << "You entered: " << strValue << std::endl;
}
```

Используя эту функцию, мы можем заставить пользователя ввести свой номер телефона точно по заданному нами шаблону. Но это не панацея на все случаи жизни.

Числовая валидация

При работе с числовым вводом очевидным путем развития событий является использование оператора извлечения для конвертации пользовательского ввода в числовой тип. Проверяя failbit, мы можем сказать, ввел ли пользователь число или нет. Например:

```
#include <iostream>

int main()
{
    int nAge;

    while (1)
    {
        std::cout << "Enter your age: ";
        std::cin >> nAge;

        if (std::cin.fail()) // если никакого извлечения не произошло
        {
            std::cin.clear(); // то сбрасываем все текущие флаги состояния и устанавливаем goodbit, чтобы иметь возм
            std::cin.ignore(32767, '\n'); // очищаем поток от мусора
            continue; // просим пользователя ввести свой возраст еще раз
        }

        if (nAge <= 0) // убеждаемся, что nAge является положительным числом
            continue;

        break;
```

```
    }

    std::cout << "You entered: " << nAge << std::endl;
}
```

Если пользователь ввел число, то `cin.fail()` будет `false`, выполнится **оператор break**, и мы выйдем из **цикла while**. Если же пользователь ввел букву, то `cin.fail()` будет `true`, и пользователю снова будет предложено ввести свой возраст.

Однако, есть один нюанс: если пользователь введет строку, которая начинается с цифр, но затем содержит буквы (например, `53qwerty74`), то первые цифры (`53`) будут извлечены в `nAge`, а остаток строки (`qwerty74`) останется во входном потоке, и failbit при этом НЕ будет установлен. Это грозит наличием мусора во входном потоке при следующем извлечении.

Давайте решим эту проблему:

```
#include <iostream>

int main()
{
    int nAge;

    while (1)
    {
        std::cout << "Enter your age: ";
        std::cin >> nAge;

        if (std::cin.fail()) // если никакого извлечения не произошло
        {
            std::cin.clear(); // то сбрасываем все текущие флаги состояния и устанавливаем goodbit, чтобы иметь возм
            std::cin.ignore(32767, '\n'); // очищаем поток от мусора
            continue; // просим пользователя ввести свой возраст еще раз
```

```
        }

        std::cin.ignore(32767, '\n'); // очищаем весь мусор, который остался в потоке после извлечения
        if (std::cin.gcount() > 1) // если мы очистили более одного символа
            continue; // то этот ввод считается некорректным, и мы просим пользователя ввести свой возраст еще раз

        if (nAge <= 0) // убеждаемся, что nAge является положительным числом
            continue;

        break;
    }

    std::cout << "You entered: " << nAge << std::endl;
}
```

Числовая валидация с помощью строки

Вышеприведенный пример потребовал немало усилий, чтобы получить одно простое значение! Другой способ обработки числового ввода заключается в том, чтобы прочитать пользовательский ввод как строку, обработать его как строку и, если он пройдет проверку, конвертировать (эту строку) в числовой тип. Например:

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

int main()
{
    int nAge;

    while (1)
    {
        std::cout << "Enter your age: ";
        std::string strAge;
```

```
        std::cin >> strAge;

        // Убеждаемся, что каждый символ является цифрой
        bool bValid = true;
        for (unsigned int nIndex = 0; nIndex < strAge.length(); nIndex++)
            if (!isdigit(strAge[nIndex]))
            {
                bValid = false;
                break;
            }
        if (!bValid)
            continue;

        // На данный момент у нас есть что-то, что мы можем конвертировать в число,
        // поэтому мы используем stringstream для выполнения конвертации
        std::stringstream strStream;
        strStream << strAge;
        strStream >> nAge;

        if (nAge <= 0) // убеждаемся, что nAge является положительным числом
            continue;

        break;
    }

    std::cout << "You entered: " << nAge << std::endl;
}
```

Будет ли этот вариант более эффективным, нежели прямое числовое извлечение, зависит от ваших параметров валидации и ограничений.

Как вы можете видеть, валидация пользовательского ввода в языке C++ занимает не так уж и мало времени и усилий. К счастью, многие подобные задачи (например, выполнение числовой валидации с помощью строк) можно легко

превратить в функции, которые затем можно будет повторно использовать в других программах.

Оценить статью:

(**97** оценок, среднее: **4,88** из 5)

Поделиться в социальных сетях:

Урок №210. Потоковые классы и Строки

Урок №212. Базовый файловый ввод и вывод

Комментариев: 6

Максим

Урок №212. Базовый файловый ввод и вывод

Юрий | Уроки по C++ | Обновл. 15 Сен 2021 | 68401

Работа файлового ввода/вывода в языке C++ почти аналогична работе обычных **потоков ввода/вывода** (но с небольшими нюансами).

Оглавление:

Классы файлового ввода/вывода

Есть **три основных класса файлового ввода/вывода в языке C++**:

- **ifstream** (является **дочерним** классу **istream**);
- **ofstream** (является дочерним классу **ostream**);
- **fstream** (является дочерним классу `iostream`).

С помощью этих классов можно выполнить однонаправленный файловый ввод, однонаправленный файловый вывод и двунаправленный файловый ввод/вывод. Для их использования нужно всего лишь подключить **заголовочный файл** fstream.

В отличие от потоков cout, cin, cerr и clog, которые сразу же можно использовать, файловые потоки должны быть явно установлены программистом. То есть, чтобы открыть файл для чтения и/или записи, нужно создать объект соответствующего класса файлового ввода/вывода, указав имя файла в качестве параметра. Затем, с помощью оператора вставки (`<<`) или оператора извлечения (`>>`), можно записывать данные в файл или считывать содержимое файла. После проделывания данных действий нужно закрыть файл — явно вызвать **метод close()** или

просто позволить файловой переменной ввода/вывода выйти из области видимости (**деструктор** файлового класса ввода/вывода закроет этот файл автоматически вместо нас).

Файловый вывод

Для записи в файл используется класс `ofstream`. Например:

```
#include <iostream>
#include <fstream>
#include <cstdlib> // для использования функции exit()

int main()
{
    using namespace std;

    // Класс ofstream используется для записи данных в файл.
    // Создаем файл SomeText.txt
    ofstream outf("SomeText.txt");

    // Если мы не можем открыть этот файл для записи данных,
    if (!outf)
    {
        // то выводим сообщение об ошибке и выполняем функцию exit()
        cerr << "Uh oh, SomeText.txt could not be opened for writing!" << endl;
        exit(1);
    }

    // Записываем в файл следующие две строки
    outf << "See line #1!" << endl;
    outf << "See line #2!" << endl;

    return 0;
```

```
26
27      // Когда outf выйдет из области видимости, то деструктор класса ofstream автоматически закроет наш файл
28  }
```

Если вы загляните в каталог вашего проекта (*ПКМ по вкладке с названием вашего файла .cpp в Visual Studio > «Открыть содержащую папку»*), то увидите файл с именем SomeText.txt, в котором находятся следующие строки:

```
See line #1!
See line #2!
```

Обратите внимание, мы также можем использовать **метод put()** для записи одного символа в файл.

Файловый ввод

Теперь мы попытаемся прочитать содержимое файла, который создали в предыдущем примере. Обратите внимание, `ifstream` возвратит `0`, если мы достигли конца файла (это удобно для определения «длины» содержимого файла). Например:

```
1   #include <iostream>
2   #include <fstream>
3   #include <string>
4   #include <cstdlib> // для использования функции exit()
5
6   int main()
7   {
8       using namespace std;
9
10      // ifstream используется для чтения содержимого файла.
11      // Попытаемся прочитать содержимое файла SomeText.txt
12      ifstream inf("SomeText.txt");
13
14      // Если мы не можем открыть этот файл для чтения его содержимого,
```

```
    if (!inf)
    {
        // то выводим следующее сообщение об ошибке и выполняем функцию exit()
        cerr << "Uh oh, SomeText.txt could not be opened for reading!" << endl;
        exit(1);
    }

    // Пока есть данные, которые мы можем прочитать,
    while (inf)
    {
        // то перемещаем эти данные в строку, которую затем выводим на экран
        string strInput;
        inf >> strInput;
        cout << strInput << endl;
    }

    return 0;

    // Когда inf выйдет из области видимости, то деструктор класса ifstream автоматически закроет наш файл
}
```

Результат выполнения программы:

```
See
line
#1!
See
line
#2!
```

Хм, это не совсем то, что мы хотели. Как мы уже узнали на предыдущих уроках, оператор извлечения работает с «отформатированными данными», т.е. он игнорирует все пробелы, символы табуляции и символ новой строки.

Чтобы прочитать всё содержимое как есть, без его разбивки на части (как в примере, приведенном выше), нам нужно использовать **метод getline()**:

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib> // для использования функции exit()

int main()
{
    using namespace std;

    // ifstream используется для чтения содержимого файлов.
    // Мы попытаемся прочитать содержимое файла SomeText.txt
    ifstream inf("SomeText.txt");

    // Если мы не можем открыть файл для чтения его содержимого,
    if (!inf)
    {
        // то выводим следующее сообщение об ошибке и выполняем функцию exit()
        cerr << "Uh oh, SomeText.txt could not be opened for reading!" << endl;
        exit(1);
    }

    // Пока есть, что читать,
    while (inf)
    {
        // то перемещаем то, что можем прочитать, в строку, а затем выводим эту строку на экран
        string strInput;
        getline(inf, strInput);
        cout << strInput << endl;
    }

```

```
    return 0;

    // Когда inf выйдет из области видимости, то деструктор класса ifstream автоматически закроет наш файл
}
```

Результат выполнения программы:

```
See line #1!
See line #2!
```

Буферизованный вывод

Вывод в языке C++ может быть буферизован. Это означает, что всё, что выводится в файловый поток, не может сразу же быть записанным на диск (в конкретный файл). Это сделано, в первую очередь, по соображениям производительности. Когда данные буфера записываются на диск, то это называется **очисткой буфера**. Одним из способов очистки буфера является закрытие файла. В таком случае всё содержимое буфера будет перемещено на диск, а затем файл будет закрыт.

Буферизация вывода обычно не является проблемой, но при определенных обстоятельствах она может вызвать проблемы у неосторожных новичков. Например, когда в буфере хранятся данные, а программа преждевременно завершает свое выполнение (либо в результате сбоя, либо путем вызова **функции exit()**). В таких случаях деструкторы классов файлового ввода/вывода не выполняются, файлы никогда не закрываются, буферы не очищаются и наши данные теряются навсегда. Вот почему хорошей идеей является явное закрытие всех открытых файлов перед вызовом функции exit().

Также буфер можно очистить вручную, используя **метод ostream::flush()** или отправив **std::flush** в выходной поток. Любой из этих способов может быть полезен для обеспечения немедленной записи содержимого буфера на диск в случае сбоя программы.

> ***Интересный нюанс*:** Поскольку `std::endl;` также очищает выходной поток, то его чрезмерное использование (приводящее к ненужным очисткам буфера) может повлиять на производительность программы (так как очистка буфера в некоторых случаях может быть затратной операцией). По этой причине программисты, которые заботятся о производительности своего кода, часто используют `\n` вместо `std::endl` для вставки символа новой строки в выходной поток, дабы избежать ненужной очистки буфера.

Режимы открытия файлов

Что произойдет, если мы попытаемся записать данные в уже существующий файл? Повторный запуск вышеприведенной программы (самая первая) показывает, что исходный файл полностью перезаписывается при повторном запуске программы. А что, если нам нужно добавить данные в конец файла? Оказывается, **конструкторы** файлового потока принимают необязательный второй параметр, который позволяет указать программисту способ открытия файла. В качестве этого параметра можно передавать **следующие флаги** (которые находятся в классе `ios`):

- **app** — открывает файл в режиме добавления;
- **ate** — переходит в конец файла перед чтением/записью;
- **binary** — открывает файл в бинарном режиме (вместо текстового режима);
- **in** — открывает файл в режиме чтения (по умолчанию для `ifstream`);
- **out** — открывает файл в режиме записи (по умолчанию для `ofstream`);
- **trunc** — удаляет файл, если он уже существует.

Можно указать сразу несколько флагов путем использования **побитового ИЛИ (|)**.

- `ifstream` по умолчанию работает в режиме ios::in;
- `ofstream` по умолчанию работает в режиме ios::out;
- `fstream` по умолчанию работает в режиме ios::in ИЛИ ios::out, что означает, что вы можете выполнять как чтение содержимого файла, так и запись данных в файл.

Теперь давайте напишем программу, которая добавит две строки в ранее созданный нами файл SomeText.txt:

```
#include <iostream>
#include <cstdlib> // для использования функции exit()
#include <fstream>

int main()
{
    using namespace std;

    // Передаем флаг ios:app, чтобы сообщить fstream, что мы собираемся добавить свои данные к уже существующим данн
    // Мы не собираемся перезаписывать файл.
    // Нам не нужно передавать флаг ios::out, поскольку ofstream по умолчанию работает в режиме ios::out
    ofstream outf("SomeText.txt", ios::app);

    // Если мы не можем открыть файл для записи данных,
    if (!outf)
    {
        // то выводим следующее сообщение об ошибке и выполняем функцию exit()
        cerr << "Uh oh, SomeText.txt could not be opened for writing!" << endl;
        exit(1);
    }

    outf << "See line #3!" << endl;
    outf << "See line #4!" << endl;

```

```
    return 0;

    // Когда outf выйдет из области видимости, то деструктор класса ofstream автоматически закроет наш файл
}
```

Теперь, если мы посмотрим содержимое SomeText.txt (запустим одну из вышеприведенных программ для чтения файла или откроем этот файл в каталоге проекта), то увидим следующее:

```
See line #1!
See line #2!
See line #3!
See line #4!
```

Явное открытие файлов с помощью функции open()

Точно так же, как мы явно закрываем файл с помощью метода close(), мы можем явно открывать файл с помощью **функции open()**. Функция open() работает аналогично конструкторам класса файлового ввода/вывода: принимает имя файла и режим (необязательно), в котором нужно открыть файл, в качестве параметров. Например:

```
#include <fstream>

int main()
{
    using namespace std;

    ofstream outf("SomeText.txt");
    outf << "See line #1!" << endl;
    outf << "See line #2!" << endl;
    outf.close(); // явно закрываем файл

    // Упс, мы кое-что забыли сделать
```

```
    outf.open("SomeText.txt", ios::app);
    outf << "See line #3!" << endl;
    outf.close();

    return 0;

    // Когда outf выйдет из области видимости, то деструктор класса ofstream автоматически закроет наш файл
}
```

Результат:

```
See line #1!
See line #2!
See line #3!
```

На этом всё! На следующем уроке мы рассмотрим рандомный файловый ввод/вывод.

Оценить статью:

(**161** оценок, среднее: **4,92** из 5)

Поделиться в социальных сетях:

Урок №211. Состояния потока и валидация пользовательского ввода

Урок №213. Рандомный файловый ввод и вывод

Добавить комментарий

Ваш E-mail не будет опубликован. Обязательные поля помечены *

Имя *

Email *

Комментарий *

Урок №213. Рандомный файловый ввод и вывод

Юрий | Уроки по C++ | Обновл. 15 Сен 2021 | 33900 | 7

На этом уроке мы рассмотрим реализацию рандомного файлового ввода/вывода в языке C++.

Оглавление:

Файловый указатель

Каждый **класс файлового ввода/вывода** содержит **файловый указатель**, который используется для отслеживания текущей позиции чтения/записи данных в файле. Любая запись в файл или чтение содержимого файла происходит в текущем местоположении файлового указателя. По умолчанию, при открытии файла для чтения или записи, файловый указатель находится в самом начале этого файла. Однако, если файл открывается в режиме добавления, то файловый указатель перемещается в конец файла, чтобы пользователь имел возможность добавить данные в файл, а не перезаписать его.

Рандомный доступ к файлам с помощью функций seekg() и seekp()

До этого момента мы осуществляли последовательный доступ к файлам, т.е. выполняли чтение/запись файла по порядку. Тем не менее, мы можем выполнить и **произвольный** (**рандомный**) **доступ** к файлу (т.е. перемещаться по файлу, как захотим). Это может быть полезно, когда файл имеет обширное содержимое, а нам нужна всего лишь небольшая конкретная запись из всего этого. Вместо последовательного доступа (когда мы переходим до нужной записи начиная с самого начала файла), мы можем осуществить непосредственный доступ к этой записи.

Рандомный доступ к файлу осуществляется путем манипулирования файловым указателем с помощью **функции seekg()** (окончание *«**g**«* = *«**g**et»*, т.е. *«получить/достать»*) — для ввода, и **функции seekp()** (окончание *«**p**«* = *«**p**ut»* (т.е. *«положить/поместить»*) — для вывода.

Функции seekg() и seekp() принимают следующие **два параметра**:

- **первый параметр** — это смещение на которое следует переместить файловый указатель (измеряется в байтах);

- **второй параметр** — это флаг **ios**, который обозначает место, от которого следует отталкиваться при выполнении смещения.

Флаги ios, которые принимают функции seekg() и seekp() в качестве второго параметра:

- **beg** — смещение относительно начала файла (по умолчанию);

- **cur** — смещение относительно текущего местоположения файлового указателя;

- **end** — смещение относительно конца файла.

Положительное смещение означает перемещение файлового указателя в сторону конца файла, тогда как отрицательное смещение означает перемещение файлового указателя в сторону начала файла. Например:

```
inf.seekg(15, ios::cur); // перемещаемся вперед на 15 байт относительно текущего местоположения файлового указателя
inf.seekg(-17, ios::cur); // перемещаемся назад на 17 байт относительно текущего местоположения файлового указателя
inf.seekg(24, ios::beg); // перемещаемся к 24-му байту относительно начала файла
inf.seekg(25); // перемещаемся к 25-му байту файла
inf.seekg(-27, ios::end); // перемещаемся к 27-му байту от конца файла
```

Перемещение в начало или в конец файла:

```
inf.seekg(0, ios::beg); // перемещаемся в начало файла
inf.seekg(0, ios::end); // перемещаемся в конец файла
```

Теперь давайте совместим функцию seekg() с файлом SomeText.txt, рассмотренном на предыдущем уроке.

Содержимое файла SomeText.txt:

```
See line #1!
See line #2!
See line #3!
See line #4!
```

Код программы:

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib> // для использования функции exit()

int main()
{
    using namespace std;

    ifstream inf("SomeText.txt");

    // Если мы не можем открыть файл для чтения его содержимого,
    if (!inf)
    {
        // то выводим следующую ошибку и выполняем функцию exit()
        cerr << "Uh oh, SomeText.txt could not be opened for reading!" << endl;
        exit(1);
    }

    string strData;

    inf.seekg(6); // перемещаемся к 6-му символу первой строки
    // Получаем остальную часть строки и выводим её на экран
    getline(inf, strData);
    cout << strData << endl;

```

```
    inf.seekg(9, ios::cur); // перемещаемся вперед на 9 байт относительно текущего местоположения файлового указател
    // Получаем остальную часть строки и выводим её на экран
    getline(inf, strData);
    cout << strData << endl;

    inf.seekg(-14, ios::end); // перемещаемся на 14 байт назад относительно конца файла
    // Получаем остальную часть строки и выводим её на экран
    getline(inf, strData);
    cout << strData << endl;

    return 0;
}
```

Результат выполнения программы:

```
ne #1!
#2!
See line #4!
```

> ***Примечание*:** В некоторых компиляторах реализация функций seekg() и tellg() при использовании с текстовыми файлами может иметь **баги** (из-за буферизации данных). Если ваш компилятор является одним из таковых (ваш результат будет отличаться от вышеприведенного результата), то вы можете попробовать открыть файл в бинарном режиме:
>
> ```
> ifstream inf("SomeText.txt", ifstream::binary);
> ```

Есть еще две другие полезные функции — **tellg()** и **tellp()**, которые возвращают абсолютную позицию файлового указателя. Это полезно при определении размера файла:

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

```
3
4 int main()
5 {
6     std::ifstream inf("SomeText.txt");
7     inf.seekg(0, std::ios::end); // перемещаемся в конец файла
8     std::cout << inf.tellg();
9 }
```

Результат:

```
56
```

Это мы получили размер файла SomeText.txt в байтах.

Одновременное чтение и запись в файл с помощью fstream

Класс `fstream` (почти) способен одновременно читать содержимое файла и записывать данные в него! Нюанс заключается в том, что вы не можете произвольно переключаться между чтением и записью файла. Как только начнется чтение или запись файла, то единственным способом переключиться между чтением или записью будет выполнение операции, которая изменит текущее положение файлового указателя (например, поиск данных). Если вы не хотите перемещать файловый указатель (потому что он уже находится в нужном месте), то вы можете просто выполнить поиск текущих данных (на которые указывает файловый указатель):

```
1 // Предположим, что iofile является объектом класса fstream
2 iofile.seekg(iofile.tellg(), ios::beg); // перемещаемся к текущей позиции файлового указателя
```

Теперь давайте напишем программу, которая откроет файл, прочитает его содержимое и заменит все найденные гласные буквы на символ `#`:

```
1 #include <iostream>
2 #include <fstream>
3 #include <cstdlib> // для использования функции exit()
```

```
4
5  int main()
6  {
7      using namespace std;
8
9      // Мы должны указать как in, так и out, поскольку используем fstream
10     fstream iofile("SomeText.txt", ios::in | ios::out);
11
12     // Если мы не можем открыть iofile,
13     if (!iofile)
14     {
15         // то выводим сообщение об ошибке и выполняем функцию exit()
16         cerr << "Uh oh, SomeText.txt could not be opened!" << endl;
17         exit(1);
18     }
19
20     char chChar;
21
22     // Пока есть данные для обработки
23     while (iofile.get(chChar))
24     {
25         switch (chChar)
26         {
27         // Если мы нашли гласную букву,
28         case 'a':
29         case 'e':
30         case 'i':
31         case 'o':
32         case 'u':
33         case 'A':
34         case 'E':
35         case 'I':
36         case 'O':
37         case 'U':
```

```
38
39              // то перемещаемся на один символ назад относительно текущего местоположения файлового указателя
40              iofile.seekg(-1, ios::cur);
41
42              // Поскольку мы выполнили операцию поиска, то теперь можем переключиться на запись данных в файл.
43              // Заменим найденную гласную букву символом #
44              iofile << '#';
45
46              // Теперь нам нужно вернуться назад в режим чтения файла.
47              // Выполняем функцию seekg() к текущей позиции
48              iofile.seekg(iofile.tellg(), ios::beg);
49
50              break;
51          }
52      }
53
54      return 0;
55  }
```

Результат выполнения программы (содержимое файла SomeText.txt):

```
S## l#n# #1!
S## l#n# #2!
S## l#n# #3!
S## l#n# #4!
```

Другие полезные методы классов файлового ввода/вывода в языке C++:

- **remove()** — удаляет файл;
- **is_open()** — возвращает `true`, если поток в данный момент открыт, и `false` — если закрыт.

Предупреждение о записи указателей в файлы

Хотя записывать переменные в файл достаточно просто, всё становится немного сложнее, когда мы начинаем работать с **указателями**. Как мы уже знаем, указатель содержит лишь адрес переменной, на которую он указывает. Хотя эти адреса можно записывать в файл и считывать их из файла — это чревато неприятностями, так как адрес одной и той же переменной может отличаться при каждом повторном запуске программы. Следовательно, хотя переменная могла находиться по адресу `003AFCD4`, когда вы записывали этот адрес на диск (в какой-нибудь файл), при повторном запуске программы она уже может находиться по другому адресу!

Например, предположим, что у нас есть переменная `someValue` типа int, которая находится по адресу `003AFCD4`. Мы присваиваем `someValue` значение `7`. Затем объявляем указатель `*pnValue`, который указывает на `someValue` (адрес `someValue` — `003AFCD4`). Мы записываем значение `7` и значение `pnValue` (`003AFCD4`) в какой-нибудь файл.

Через несколько недель мы снова запускаем эту программу и пытаемся извлечь значения из файла. Мы извлекаем значение `7` в переменную `someValue`, которая в текущей программе уже находится по адресу `0034FD90`. Дальше мы извлекаем адрес `003AFCD4` в указатель `*pnValue`. Поскольку `pnValue` указывает на `003AFCD4`, а `someValue` находится по адресу `0034FD90`, то `pnValue` больше не указывает на `someValue`, и попытка доступа к значению адреса, который хранит `pnValue`, приведет к неприятностям.

Правило: Не сохраняйте адреса переменных в файлах. Переменные, которые изначально были по одним адресам, при повторном запуске программы могут находиться уже по другим адресам.

Поделиться в социальных сетях:

Урок №212. Базовый файловый ввод и вывод

Конфигурация компилятора: Расширения компилятора

Комментариев: 7

Карен

9 мая 2021 в 06:44

А как настроить интерфейс VS как на примерах? Имеется ввиду цветовая тема, шрифт именно такой, чередующиеся цвета для фона строк?

Ответить