Потоки ввода-вывода

Иерархия потоковых классов. Операции ввода-вывода для стандартных типов. Организация ввода-вывода для пользовательских типов. Работа с файлами и буферами в памяти. Средства форматирование вывода. Манипуляторы.

[Старая и новая системы ввода-вывода](#_gjdgxs)

[Потоки в С++](#_30j0zll)

[Потоковые классы](#_1fob9te)

[Функциональность класса istream](#_3znysh7)

[Функциональность класса ostream](#_2et92p0)

[Потоковые классы и строки](#_efnlbkbg1j9i)

[Перегрузка оператора вывода](#_3dy6vkm)

[Перегрузка оператора ввода](#_1t3h5sf)

[Написание игры Blackjack](#_cyec4m4l55pk)

[Практическое задание](#_4d34og8)

[Дополнительные материалы](#_y937sk8fclye)

[Используемая литература](#_uvp6qax5r1ok)

# Старая и новая системы ввода-вывода

Есть две версии объектно-ориентированной библиотеки ввода-вывода С++. Старая библиотека поддерживается заголовочным файлом **<iostream.h>**. Новая —заголовком **<iostream>**. Библиотеки с точки зрения программиста в целом одинаковые, просто новая — модернизированная и улучшенная версия старой. Большая часть различий лежит под поверхностью и определяется тем, как библиотека реализована.

Новая библиотека является надмножеством старой, и все программы, написанные для старой библиотеки, будут компилироваться без существенных изменений, если использовать новую библиотеку. Она находится в пространстве имен **std**, в то время как старая — в глобальном пространстве имен.

# Потоки в С++

Система ввода-вывода С++ оперирует над потоками. Поток является абстрактным объектом, который либо создает, либо поглощает информацию. Поток связывается с физическим устройством посредством системы ввода-вывода. Все потоки ведут себя одинаково, одни и те же функции и операторы ввода-вывода применимы практически ко всем типам устройств. Например, метод, с помощью которого выводятся данные на экран, можно использовать для записей данных на диск или печати на принтере.

В наиболее общей форме поток — это логический интерфейс с файлом. Согласно определению в С++, термин «файл» может относиться к клавиатуре, порту, файлу и так далее. Поток в данном случае обеспечивает единообразный интерфейс.

Чтение данных из потока называется извлечением, вывод в поток — помещением или включением. Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен. Обмен с потоком для увеличения скорости передачи данных происходит, как правило, через специальную область данных — буфер.

По направлению обмена потоки можно разделить на входные, где данные вводятся в память, и двунаправленные, допускающие как извлечение, так и включение.

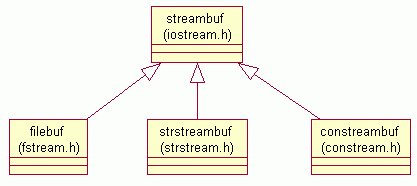
По виду устройств, с которыми работает поток, можно выделить:

* стандартные потоки — предназначены для передачи данных от клавиатуры и на экран;
* файловые потоки — для обмена информацией с файлами на внешних устройствах;
* строковые потоки — для работы с массивами символов.

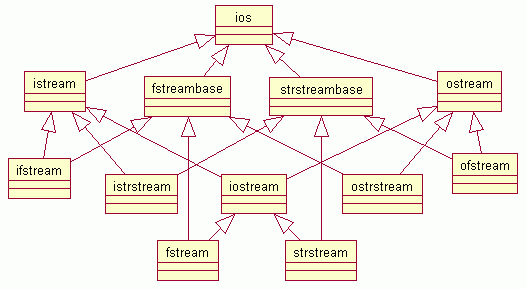
# Потоковые классы

Библиотека потоковых классов С++ построена на основе двух базовых классов: **ios** и **streambuf** .

Класс **streambuf** обеспечивает организацию и взаимосвязь буферов ввода-вывода, размещаемых в памяти, с физическими устройствами ввода-вывода. Методы и данные класса **streambuf** программист явно обычно не использует. Этот класс нужен другим классам библиотеки ввода-вывода. Он доступен и программисту для создания новых классов на основе уже существующих.



Класс **ios** содержит средства для форматированного ввода-вывода и проверки ошибок.



**Стандартные потоки** (**istream**, **ostream**, **iostream**) служат для работы с терминалом. Строковые потоки (**istrstream**, **ostrstream**, **strstream**) служат для ввода-вывода из строковых буферов, размещенных в памяти. Файловые потоки (**ifstream**, **ofstream**, **fstream** ) служат для работы с файлами.

Рассмотрим их более подробно:

* **ios** — базовый потоковый класс;
* **streambuf** — буферизация потоков;
* **istream** — потоки ввода;
* **ostream** — потоки вывода;
* **iostream** — двунаправленные потоки;
* **istrstream** — строковые потоки ввода;
* **ostrstream** — строковые потоки вывода;
* **strstream** — двунаправленные строковые потоки;
* **ifstream** — файловые потоки ввода;
* **ofstream** — файловые потоки вывода;
* **fstream** — двунаправленные файловые потоки.

Хотя класс **ios** является дочерним классу **ios\_base**. Часто именно он будет наиболее родительским классом, с которым вы будете работать напрямую. Класс **ios** определяет много разных вещей, которые являются общими для потоков ввода/вывода.

**Класс istream** используется для работы с входными потоками. **Оператор извлечения** **>>** используется для получения 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, 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.

**Стандартный поток** — это предварительно подключенный поток, который предоставляется программе ее окружением. C++ поставляется с четырьмя предварительно определенными стандартными объектами потоков, которые вы можете использовать.

Следующие объекты-потоки заранее определены и открыты в программе перед вызовом функции **main**:

| extern istream cin; *// Стандартный поток ввода с клавиатуры*  extern ostream cout; *// Стандартный поток вывода на экран*  extern ostream cerr; *// Стандартный небуферизованный поток вывода сообщений об ошибках (экран)*  extern ostream clog; *// Стандартный буферизованный поток вывода*  *// сообщений об ошибках (экран)* |
| --- |

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

Приведем пример их использования:

| #include <iostream>  #include <cstdlib> *// для exit()*  using namespace std;  int main()  {  *// Сначала мы используем оператор вставки с объектом cout для вывода текста на монитор*  cout << "Enter your age: " << endl;    *// Затем — оператор извлечения с объектом cin для получения пользовательского ввода*  int age;  cin >> age;  if (age <= 0)  {  *// В этом случае мы используем оператор вставки с объектом cerr для вывода сообщения об ошибке*  cerr << "Oops, you entered an invalid age!" << endl;  exit(1);  }  *// А здесь мы используем оператор вставки с объектом cout для вывода результата*  cout << "You entered " << age << " years old" << endl;  return 0;  } |
| --- |

Потоковые классы, их методы и данные становятся доступными в программе, если в нее включен нужный заголовочный файл:

* **iostream.h** — для **ios**, **ostream**, **istream** .
* **strstream.h** — для **strstream**, **istrstream**, **ostrstream** .
* **fstream.h**— для **fstream**, **ifstream**, **ofstream**.

# Функциональность класса istream

Библиотека **iostream** довольно сложная, поэтому мы не сможем охватить ее полностью на уроках. Рассмотрим ее основную функциональность. В этом уроке разберемся с классом **istream**.

Одной из наиболее распространенных проблем при считывании строк из входного потока является переполнение — например, если мы выделили символьный массив на 10 символов, а пользователь введет 20, произойдет переполнение. Чтобы избежать этой ситуации, используют **манипуляторы**.  Это объекты, которые применяются 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. 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>  using namespace std;  int main()  {  char name[12];  cin >> setw(12) >> name;  } |
| --- |

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.

Важный момент — оператор извлечения работает с «отформатированными» данными, то есть он игнорирует все пробелы, символы табуляции и новой строки.

Часто пользовательский ввод все же нужен со всеми его пробелами. Для этого класс **istream** предоставляет множество функций. Одной из наиболее полезных является **функция** **get()**, которая извлекает символ из входного потока.

| #include <iostream>  using namespace std;  int main()  {  char fio;  while (cin.get(fio))  cout << fio;  return 0;  } |
| --- |

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

| #include <iostream>  using namespace std;  int main()  {  char fio[30];  cin.get(fio, 30);  cout << fio << endl;  return 0;  } |
| --- |

Один важный нюанс: **get()** не считывает символ новой строки! Для решения этой проблемы класс **istream** предоставляет **функцию** **getline()**, которая работает так же, как **get()**, но при этом может считывать символы новой строки.

Если вам нужно узнать количество символов, извлеченных последним **getline()**, используйте функцию **gcount()**. Для вывода количества символов после использования **cin.getline()** можно применять следующую строчку:

**cout << cin.gcount();**

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

| #include <iostream>  #include <string>  using namespace std;  int main()  {  string fio;  getline(cin, fio);  cout << fio << endl;  return 0;  } |
| --- |

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

| Метод | Описание |
| --- | --- |
| ignore() | Метод извлекает один символ из потока **istream** и игнорирует его |
| ignore(streamsize  count) | Метод извлекает **count**-символ из потока **istream** и игнорирует его |
| ignore(streamsize  count, char  delim) | Метод извлекает **count**-символ из потока **istream** и игнорирует его. Метод прекращает работу, если в потоке встречается символ **delim** |
| peek() | Метод возвращает следующий символ потока без реального чтения его из потока **istream** |
| unget() | Метод помещает в поток последний считанный символ таким образом, что его можно прочитать следующей операцией чтения |

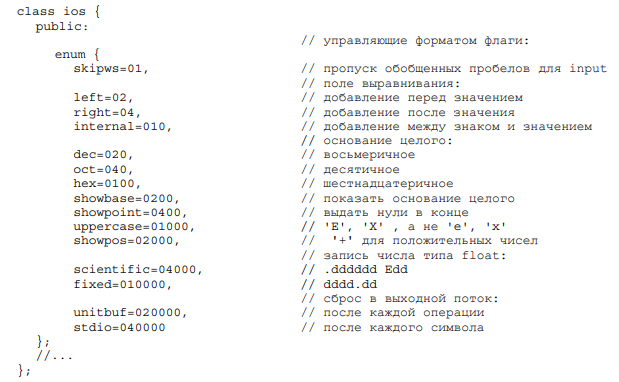
# Функциональность класса ostream

Оператор вставки (вывода) **<<** используется для помещения информации в выходной поток. Классы **istream** и **ostream** — дочерние 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 включать/выключать;
* манипуляторы — это объекты, которые помещаются в поток и влияют на способ ввода/вывода данных.

В классе **ios** содержится состояние формата, которое управляется функциями **flags()** и **setf()**. По сути, эти функции нужны, чтобы установить или отменить следующие флаги:



Конкретные значения флагов зависят от реализации и даны здесь только для того, чтобы избежать синтаксически неверных конструкций. Определение интерфейса как набора флагов и операций для их установки или отмены — это прошедший проверку временем, хотя и устаревший прием.

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

| #include <iostream>  using namespace std;  int main()  {  cout.setf(ios::showpos); *// включаем флаг showpos*  cout << 30;  } |
| --- |

Результат работы программы: **+30**.

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

Многие флаги принадлежат к определенным **группам форматирования** — группам флагов, выполняющих аналогичные (иногда взаимоисключающие) параметры форматирования вывода.

*Флаги группы форматирования* ***basefield****:*

* **oct** («**octal**» = «восьмеричный») — восьмеричная система счисления;
* **dec** («**decimal**» = «десятичный») — десятичная система счисления;
* **hex** («**hexadecimal**» = «шестнадцатеричный») — шестнадцатеричная система счисления.

Эти флаги управляют выводом целочисленных значений. По умолчанию установлен флаг **dec**, то есть значения выводятся в десятичной системе счисления. Чтобы число вывести в шестнадцатеричной системе, нужно сначала отключить флаг **dec**, а затем включить флаг **hex**. Есть и другой способ: можно функции **setf()** передать два параметра: первый — это флаг, который нужно включить/выключить, второй — группа форматирования, к которой принадлежит флаг.

Запись будет выглядеть так:

| cout.setf(ios::hex, ios::basefield); |
| --- |

Помимо флагов можно использовать и манипуляторы:

| cout << hex << 30; |
| --- |

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

Рассмотрим список наиболее полезных флагов, манипуляторов и методов. Флаги находятся в классе **ios**, манипуляторы — в пространстве имен **std**, а методы — в классе **ostream**.

| **Флаг** | **Манипулятор** |
| --- | --- |
| **boolalpha** — если включен, то логические значения выводятся как «true/false». Если выключен, то как «0/1». | **boolalpha** — логические значения выводятся как «true/false».  **noboolalpha** — логические значения выводятся как «0/1». |
| **showpos** — если включен, то к положительным числам прибавляется знак +. | **showpos** — к положительным числам прибавляется знак +.  **noshowpos** — к положительным числам не прибавляется знак +. |
| **uppercase** — если включен, то используются заглавные буквы. | **uppercase** — используются заглавные буквы.  **nouppercase** — используются строчные буквы. |
| **dec** — значения выводятся в десятичной системе счисления.  **hex** — значения выводятся в шестнадцатеричной системе счисления.  **oct** — значения выводятся в восьмеричной системе счисления. | **dec** — значения выводятся в десятичной системе счисления.  **hex** — значения выводятся в шестнадцатеричной системе счисления.  **oct** — значения выводятся в восьмеричной системе счисления. |
| **fixed** — используется десятичная запись чисел типа с плавающей запятой.  **scientific** — используется экспоненциальная запись чисел типа с плавающей запятой.  **showpoint** — всегда отображается десятичная точка и конечные нули для чисел типа с плавающей запятой. | **fixed** — используется десятичная запись значений.  **scientific** — используется экспоненциальная запись значений.  **showpoint** — отображается десятичная точка и конечные нули чисел типа с плавающей запятой.  **noshowpoint** — не отображаются десятичная точка и конечные нули чисел типа с плавающей запятой.  **setprecision(int)** — задаем точность для чисел типа с плавающей запятой. |
| **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. | **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. |

Попробуйте использовать данные флаги и манипуляторы.

Для использования манипуляторов необходимо подключить заголовочный файл **<iomanip>**.

# Потоковые классы и строки

В стандартной библиотеке С++ есть отдельный набор классов, которые позволяют использовать операторы вставки (<<) и извлечения (>>) со строками. Как **istream** и **ostream**, потоковые классы для строк предоставляют буфер для хранения данных. Но в отличие от **cin** и **cout**, эти потоковые классы не подключены к каналу ввода/вывода (то есть к клавиатуре, монитору и так далее).

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

* **istringstream** (дочерний классу **istream**);
* **ostringstream** (дочерний классу **ostream**);
* **stringstream** (дочерний классу **iostream**);
* **wistringstream**;
* **wostringstream**;
* **wstringstream**.

Чтобы использовать **stringstream**, нужно подключить заголовочный файл **sstream**.

Чтобы занести данные в **stringstream**, мы можем использовать оператор вставки (<<), либо функцию **str(string).** Аналогично, чтобы получить данные обратно из **stringstream**, можем использовать функцию **str()** или оператор извлечения (>>).

Можно применять операторы вставки и извлечения со строками для их конвертации в числа и наоборот.

Например, конвертация чисел в строки:

| #include <iostream>  #include <sstream> *// для stringstream*  using namespace std;  int main()  {  stringstream myString;  int nValue = 336000;  double dValue = 12.14;  myString << nValue << " " << dValue;    string strValue1, strValue2;  myString >> strValue1 >> strValue2;    cout << strValue1 << " " << strValue2 << endl;  } |
| --- |

Результат работы программы:

**336000 12.14**

А теперь конвертация (числовой) строки обратно в числа:

| #include <iostream>  #include <sstream> *// для stringstream*  using namespace std;  int main()  {  stringstream myString;  myString << "336000 12.14"; *// вставляем (числовую) строку в поток*  int nValue;  double dValue;  myString >> nValue >> dValue;  cout << nValue << " " << dValue << endl;  } |
| --- |

Результат работы программы:

**336000 12.14**

Чтобы очистить **stringstream** для повторного использования, можно применить функцию **str()** с пустой строкой: **myString.str("")**; или функцию **clear()**, сбрасывающую все флаги ошибок, которые были установлены, и возвращающую поток обратно в его прежнее безошибочное состояние.

# Перегрузка оператора вывода

Для вывода информации об объекте класса удобно использовать стандартный метод. Для этого необходимо перегрузить оператор ввода <<. Правым операндом данного оператора является объект класса, а левым операндом — объект **std::cout**, который имеет тип **std::ostream**. Перегружать оператор будем через дружественную функцию (а не через метод класса или обычную функцию):

| #include <iostream>  using namespace std;  class Date  {  private:  int m\_day, m\_month, m\_year;  public:  Date(int d=1, int m=1, int y=2019): m\_day(d), m\_month(m), m\_year(y)  { }  friend ostream& operator<< (ostream &out, const Date &date);  };  ostream& operator<< (ostream &out, const Date &date)  {  out << "Date: " << date.m\_day << ". " << date.m\_month << ". " << date.m\_year << "\n";  return out;  }  int main()  {  Date date(5, 4, 2019);  cout << date;  return 0;  } |
| --- |

Обратите внимание на тип возврата оператора. Результат возвращается по ссылке, так как по значению возврат запрещен из-за невозможности копирования **std::ostream**. Возврат по ссылке не только предотвращает копирование класса **std::ostream**, но и позволяет связать выражения вывода — например, **cout << date << endl**.

Каждый раз, когда мы хотим, чтобы перегруженные бинарные операторы были связаны таким образом, левый операнд должен быть возвращен по ссылке. Возврат левого параметра по ссылке в этом случае работает, так как он передается в функцию самим ее вызовом, и должен оставаться даже после выполнения и возврата этой функции. Так что мы можем не беспокоиться о том, что ссылаемся на что-то, что выйдет из области видимости и уничтожится после выполнения функции.

# Перегрузка оператора ввода

Перегрузки операторов ввода и вывода очень похожи. Единственное отличие состоит в том, что **std::cin** является объектом типа **std::istream**. Перегрузим оператор ввода для предыдущего примера:

| istream& operator>> (istream &in, Date &date)  {  *// обратите внимание, параметр date (объект класса Date) должен быть не константным, чтобы мы имели возможность изменить члены класса*  in >> date.m\_day;  in >> date.m\_month;  in >> date.m\_year;  return in;  } |
| --- |

Перегрузка осуществляется также через дружественную функцию, которая имеет доступ к приватным переменным класса. Для использования данного оператора в функции **main()** достаточно написать:

**Date date;**

**cin >> date;**

# Написание игры Blackjack

Напишем перегрузку оператора вывода для класса **Card**:

| *// перегружает оператор <<, чтобы получить возможность отправить*  *// объект типа Card в поток cout*  ostream& operator<<(ostream& os, const Card& aCard)  {  const string RANKS[] = { "0", "A", "2", "3", "4", "5", "6", "7", "8", "9","10", "J", "Q", "K" };  const string SUITS[] = { "c", "d", "h", "s" };    if (aCard.m\_IsFaceUp)  {  os << RANKS[aCard.m\_Rank] << SUITS[aCard.m\_Suit];  }  else  {  os << "XX";  }    return os;  } |
| --- |

Перегружать будем через дружественную функцию, поэтому в самом классе объявим ее:

| friend ostream& operator<<(ostream& os, const Card& aCard); |
| --- |

Теперь разберемся с классом **GenericPlayer**, который обобщенно представляет игрока в Blackjack. Он представляет не полноценного игрока, а общие элементы игрока-человека и игрока-компьютера.

| *// абстрактный класс*  class GenericPlayer : public Hand  {  friend ostream& operator<<(ostream& os, const GenericPlayer& aGenericPlayer);    public:  GenericPlayer(const string& name = "");    virtual ~GenericPlayer();    *// показывает, хочет ли игрок продолжать брать карты*  *// Для класса GenericPlayer функция не имеет своей реализации,*  *// т.к. для игрока и дилера это будут разные функции*  virtual bool IsHitting() const = 0;    *// возвращает значение, если у игрока перебор -*  *// сумму очков большую 21*  *// данная функция не виртуальная, т.к. имеет одинаковую реализацию*  *// для игрока и дилера*  bool IsBusted() const;    *// объявляет, что игрок имеет перебор*  *// функция одинакова как для игрока, так и для дилера*  void Bust() const;    protected:  string m\_Name;  }; |
| --- |

Конструктор данного класса принимает строку, представляющую собой имя игрока. Деструктор автоматически становится виртуальным, поскольку наследует это свойство от класса **Hand**.

| GenericPlayer::GenericPlayer(const string& name) :  m\_Name(name)  {}  GenericPlayer::~GenericPlayer()  {} |
| --- |

Функция-член **IsHitting()** показывает, хочет ли игрок взять еще одну карту. Поскольку эта функция-член не имеет реального значения для обобщенного класса **GenericPlayer**, она является чисто виртуальной. Благодаря этому класс становится абстрактным. Соответственно, в классах **Player** и **House** должны быть реализованы собственные версии этой функции.

Функция-член **IsBoosted()** показывает, есть ли у игрока перебор. Поскольку перебор у дилера и игроков одинаков — сумма очков их карт превосходит 21, — функция размещена внутри этого класса.

| bool GenericPlayer::IsBusted() const  {  return (GetTotal() > 21);  } |
| --- |

Функция-член **Bust()** объявляет, что у игрока перебор. Поскольку перебор для игроков и дилера объявляется одинаковым образом, функция-член размещена внутри данного класса.

| void GenericPlayer::Bust() const  {  cout << m\_Name << " busts.\n";  } |
| --- |

Для класса **GenericPlayer** существует перегрузка оператора вывода **<<**. Функция отображает имя игрока и его карты, а также общую сумму очков его карт.

| ostream& operator<<(ostream& os, const GenericPlayer& aGenericPlayer)  {  os << aGenericPlayer.m\_Name << ":\t";    vector<Card\*>::const\_iterator pCard;  if (!aGenericPlayer.m\_Cards.empty())  {  for (pCard = aGenericPlayer.m\_Cards.begin();  pCard != aGenericPlayer.m\_Cards.end();  ++pCard)  {  os << \*(\*pCard) << "\t";  }      if (aGenericPlayer.GetTotal() != 0)  {  cout << "(" << aGenericPlayer.GetTotal() << ")";  }  }  else  {  os << "<empty>";  }    return os;  } |
| --- |

Теперь рассмотрим класс игрока-человека **Player**. Он наследует от класса **GenericPlayer**.

| class Player : public GenericPlayer  {  public:  Player(const string& name = "");    virtual ~Player();    *// показывает, хочет ли игрок продолжать брать карты*  virtual bool IsHitting() const;    *// объявляет, что игрок победил*  void Win() const;    *// объявляет, что игрок проиграл*  void Lose() const;    *// объявляет ничью*  void Push() const;  }; |
| --- |

Этот класс реализует функцию-член **IsHitting()**, которая унаследована от класса **GenericPlayer**. Поэтому класс **Player** не является абстрактным. Класс реализует функцию-член, спрашивая у человека, хочет ли он взять еще одну карту. Если игрок вводит символ **y** или **Y** в ответ, функция-член возвращает **true**, что показывает — игрок хочет взять еще одну карту. Если же игрок вводит любой другой символ, эта функция возвращает **false**, и это означает, что игрок больше не хочет брать карту.

| bool Player::IsHitting() const  {  cout << m\_Name << ", do you want a hit? (Y/N): ";  char response;  cin >> response;  return (response == 'y' || response == 'Y');  } |
| --- |

Функции-члены **Win()**, **Lose()** и **Push()** просто объявляют, что игрок выиграл, проиграл и сыграл вничью соответственно.

| void Player::Win() const  {  cout << m\_Name << " wins.\n";  }  void Player::Lose() const  {  cout << m\_Name << " loses.\n";  }  void Player::Push() const  {  cout << m\_Name << " pushes.\n";  } |
| --- |

Класс **House** представляет дилера. Он наследует от класса **GenericPlayer**.

| class House : public GenericPlayer  {  public:  House(const string& name = "House");    virtual ~House();    *// показывает, хочет ли дилер продолжать брать карты*  virtual bool IsHitting() const;    *// переворачивает первую карту*  void FlipFirstCard();  }; |
| --- |

Это класс реализует функцию-члн **IsHitting()**, которая унаследована от класса **GenericPlayer**. Поэтому класс **House** не является абстрактным.

| bool House::IsHitting() const  {  return (GetTotal() <= 16);  } |
| --- |

Класс реализует эту функцию-член, вызывая функцию **GetTotal()**. Если возвращенное значение меньше или равно 16, функция-член возвращает значение **true**, что показывает: дилер хочет взять еще одну карту. В противном случае функция возвращает значение **false**, и это означает, что дилеру карты больше не нужны.

| bool House::IsHitting() const  {  return (GetTotal() <= 16);  } |
| --- |

Функция-член **FlipFirstCard()** переворачивает первую карту дилера. Она необходима, поскольку дилер скрывает свою первую карту в начале кона, а затем показывает ее после того, как все игроки взяли дополнительные карты.

| void House::FlipFirstCard()  {  if (!(m\_Cards.empty()))  {  m\_Cards[0]->Flip();  }  else  {  cout << "No card to flip!\n";  }  } |
| --- |

# Практическое задание

1. Создать программу, которая считывает целое число типа **int**. И поставить «защиту от дурака»: если пользователь вводит что-то кроме одного целочисленного значения, нужно вывести сообщение об ошибке и предложить ввести число еще раз. Пример неправильных введенных строк:

**rbtrb**

**nj34njkn**

**1n**

1. Создать собственный манипулятор **endll** для стандартного потока вывода, который выводит два перевода строки и сбрасывает буфер.
2. Реализовать класс **Player**, который наследует от класса **GenericPlayer**. У этого класса будет 4 метода:

* **virtual bool IsHitting() const** - реализация чисто виртуальной функции базового класса. Метод спрашивает у пользователя, нужна ли ему еще одна карта и возвращает ответ пользователя в виде **true** или **false**.
* **void Win() const** - выводит на экран имя игрока и сообщение, что он выиграл.
* **void Lose() const** - выводит на экран имя игрока и сообщение, что он проиграл.
* **void Push() const** - выводит на экран имя игрока и сообщение, что он сыграл вничью.

1. Реализовать класс **House**, который представляет дилера. Этот класс наследует от класса **GenericPlayer**. У него есть 2 метода:

* **virtual bool IsHitting() const** - метод указывает, нужна ли дилеру еще одна карта. Если у дилера не больше 16 очков, то он берет еще одну карту.
* **void FlipFirstCard()** - метод переворачивает первую карту дилера.

1. Написать перегрузку оператора вывода для класса **Card**. Если карта перевернута рубашкой вверх (мы ее не видим), вывести **ХХ**, если мы ее видим, вывести масть и номинал карты. Также для класса **GenericPlayer** написать перегрузку оператора вывода, который должен отображать имя игрока и его карты, а также общую сумму очков его карт.

# Дополнительные материалы

1. Бьерн Страуструп. Программирование. Принципы и практика использования С++.
2. Стивен Прата. Язык программирования С++. Лекции и упражнения.
3. Роберт Лафоре. Объектно-ориентированное программирование в С++.

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Поточный ввод-вывод в C++](https://prog-cpp.ru/cpp-std/).
2. Бьерн Страуструп. Программирование. Принципы и практика использования С++.
3. Ральф Джонсон, Ричард Хелм, Эрих Гамма. Приемы объектно-ориентированного программирования. Паттерны проектирования.