# *Файловый ввод-вывод. Перегрузка операторов ввода-вывода*

Для реализации файлового ввода-вывода необходимо включить в программу заголовок <fstream>. В нем определены несколько классов, включая классы ifstream, ofstream, fstream. Эти классы являются производными от классов istream, ostream.

В C++ файл открывается посредством его связывания с потоком. Имеется три потока: ввода, вывода, ввода-вывода.

Перед тем как открыть файл необходимо:

Создать поток. Для создания потока ввода, вывода, ввода-вывода необходимо объявить соответственно объект типа ifstream, ofstream, fstream.

**Пример**:

ifstream in; //ввод

ofstream out; //вывод

fstream io //ввод-вывод

Связать его с файлом. Это можно сделать, используя функцию open(). Эта функция является членом каждого из потоковых классов.

Прототипы:

void ifstream::open(const char \* имя\_файла, openmode режим=ios::in);

void ofstream::open(const char \* имя\_файла, openmode режим=ios::out | ios::trunc);

void fstream::open(const char \* имя\_файла, openmode режим=ios::in | ios::out);

Значение режим задает режим открытия файла. Оно должно быть значением типа openmode, которое является перечислением, определенным в классе ios.

Значение режим может быть одним из следующих:

ios::app, ios::ate, ios::binary; ios::in, ios::out, ios::trunc

Можно объединить два или более этих значения с помощью оператора OR.

|  |  |
| --- | --- |
| ios::app | вызывет открытие файла в режиме добавления в конец файла |
| ios::ate | задает режим поиска конца файла при его открытии |
| ios::in | задает режим открытия файла для ввода |
| ios::out | задает режим открытия файла для вывода |
| ios::binary | открытие файла в двоичном режиме (по умолчанию в текстовом) |
| ios::trunc | приводит к удалению содержимого и усечение до нулевой длины |

**Пример:**

ofstream mystream;

mystream.open("test"); // mystream.open("test", ios::in | ios::out);

Если выполнение open() завершилось с ошибкой, в булевском выражении поток будет равен false. Поэтому, часто используют проверку:

if (!mystream) {

cout<<"Файл открыть невозможно\n";

// программа обработки ошибки открытия файла

}

Однако на практике часто open() не используют, так как конструкторы соответствующих классов открывают файл автоматически.

**Пример**:

ifstream mystream("myfile");

Для закрытия файла используют функцию-член close().

**Пример**:

mystream.close();

**Пример**:

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

ofstream fout("file"); // создание файла для вывода

if (!fout) {

cout << "Файл открыть невозможно\n";

return 1;

}

fout << "Write!!!\n";

fout << 100 << ' ' << hex << 100 << endl;

fout.close();

ifstream fin("file"); // открытие файла для ввода

if (!fin) {

cout << "Файл открыть невозможно\n";

return 1;

}

char str[80];

int i, j;

fin >> str >> i >> j;

cout << str << ' ' << i << ' ' << j << endl;

fin.close();

system("pause");

return 0;

}

**Пример**:

#include <iostream>

#include <windows.h>

using namespace std;

char bufRus[256];

char\* Rus(const char \*text)

{

CharToOem(text, bufRus);

return bufRus;

}

char bufEng[256];

char\* Eng(const char \*text)

{

OemToChar(text, bufEng);

return bufEng;

}

class student {

char fio[40];

char group[5];

char n\_st[10];

public:

student() {};

student(char const \*n, char const \*g, char const \*b)

{

strcpy\_s(fio, n); strcpy\_s(group, g); strcpy\_s(n\_st, b);

}

friend ostream& operator << (ostream &stream, student &ob);

friend istream& operator >> (istream &stream, student &ob);

};

ostream &operator << (ostream &stream, student &ob)

{

stream << Rus(ob.fio) << " " << ob.group << " " << ob.n\_st << endl;

return stream;

}

istream &operator >> (istream &stream, student &ob)

{

cout << Rus("\nВведите фамилию, имя, студента: ");

stream.getline(ob.fio, 40);

strcpy\_s(ob.fio, Eng(ob.fio));

cout << Rus("\nВведите номер группы студента: ");

stream >> ob.group;

cout << Rus("\nВведите номер идентификатора студента: ");

stream >> ob.n\_st;

cout << endl;

return stream;

}

int main()

{

// setlocale(LC\_ALL, "Russian");

// SetConsoleCP(1251);

// SetConsoleOutputCP(1251);

student ob1("Иванов Иван Иванович", "115", "050100");

student ob2("Петров Петр Петрович", "116", "050115");

student ob3("Сидоров Сидор Сидорович", "118", "050125");

student ob4;

cout << ob1 << ob2 << ob3;

cin >> ob4;

cout << ob4;

return 0;

}

**Комментарий**: В программе используются несколько функций

* **Rus(cStr)** – перевод строки cStr в Oem – кодировку;
* **Eng(cStr)** – перевод строки cStr из Oem – кодировку.

Ипользование функции getline() в C++ решает проблему ввода символьной строки, в которой есть пробелы.

Функция **getline()** является членом каждого потокового класса.

Прототипы:

istream &getline(char \**buf*, streamsize *num*);

istream &getline(char \**buf*, streamsize *num*, char *delim*);

В первом случае функция **getline()** считывает из строки, на которую указывает buf, num -1 символ, если не будет обнаружен символ перехода на следующую строку или достигнут конец файла. Функция записывает нулевой символ в конец массива, символ перехода на новую строку извлекается из потока, но *не записывается* в буфер.

Во втором случае вместо символа перехода на новую строку используется символ *delim*.

**Неформатируемый двоичный ввод-вывод**

Рассмотрим основные функции.

Класс istream

|  |  |
| --- | --- |
| функция | назначение |
| get(ch) | считывание одного символа в ch (возможна работа с массивом) |
| read(str,MAX) | (для файлов). Извлекает вплоть до MAX число символов в массив str до признака конца файла (EOF) |

Класс ostream

|  |  |
| --- | --- |
| функция | назначение |
| put(ch) | вставка символа ch в поток |
| write(str,SIZE) | вставка SIZE символов из массива str в файл |

Возвращаемым значением функции get() является ссылка на поток. При считывании символа конца файла функция возвращает потоку значение false.

Функция put() записывает символ в поток и возвращает ссылку на поток.

Узнать, сколько символов было считано, можно с помощью функции gcount().

**Произвольный доступ**

В C++ произвольный доступ в системе ввода-вывода реализуется с помощью функций seekg() и seekp(), являющихся потоковыми функциями ввода и вывода.

Прототипы:

istream &seekg(off\_type смещение, seekdir задание);

ostream &seekp(off\_type смещение, seekdir задание);

off\_type – целый тип данных, определенных в классе ios. Тип seekdir – перечисление, определенное в классе ios и содержащее значение:

ios::beg поиск с начала файла

ios::cur поиск с текущей позиции в файле

ios::end поиск с конца файла

Функция seekg() устанавливает указатель считывания соответствующего файла в позицию, отстоящую на величину смещение от заданного места "задание".

Функция seekp() устанавливает указатель записи cоответствующего файла в позицию, отстоящую на величину смещение от заданного места "задание".

Определить текущую позицию каждого из двух указателей можно с помощью функций:

pos\_type tellg();

pos\_type tellp();

pos\_type – целый тип данных, определенный в классе ios и способный хранить наибольшее возможное значение указателя.

# Стандартная библиотека. Элементы библиотеки стандартных шаблонов (STL – Standard Template Library)

Стандарты языка C++ описываются в документе, состоящем из 776 страниц (ISO/IES 14882, 1998). Под библиотекой будем понимать набор программных средств многоразового использования (классы, функции, макросы и т.д.), который создается для облегчения выполнения стандартных, наиболее общих задач программирования.

Стандартной называется библиотека, состав назначение и принципы использования которой определяются стандартами Международной организации стандартизации (ISO – International Standards Organization). Стандартная библиотека представляет также среду разработки для создания новых библиотечных компонентов, хотя в большинстве случаев работа со стандартной библиотекой не подразумевает создание новых компонентов.

Стандартная библиотека C++ состоит из десяти библиотек:

1. Библиотека языковой поддержки, которая содержит типы и функции, непосредственно связанные с работой компилятора.
2. Библиотека диагностики. Содержит классы диагностики, коды и сообщения об ошибках.
3. Библиотека общего пользования.
4. Библиотека строк. Содержит средства манипулирования текстовыми строками.
5. Библиотека поддержки наборов международных и национальных символов и форматов.
6. Библиотека контейнеров. Содержит родовые контейнеры для создания коллекции объектов.
7. Библиотека итераторов. Поддерживает функции навигации по коллекциям объектов.
8. Библиотека алгоритмов. Содержит стандартные алгоритмы обработки данных.
9. Библиотека чисел. Комплексные числа, специальные типы числовых массивов и алгоритмы вычислений.
10. Библиотека ввода-вывода.

Под стандартной библиотекой шаблонов (STL) понимают набор интерфейсов и компонентов, разработанных Александром Степановым и другими сотрудниками AT&T Bell Laboratories. STL включает в себя библиотеки контейнеров, итераторов, алгоритмов и некоторые части библиотеки общего пользования, которые, в свою очередь, являются составными частями стандартной библиотеки C++. Таким образом, STL является составной частью библиотеки C++.

Ядро библиотеки стандартных шаблонов образуют три основополагающих элемента:

1. контейнеры;
2. алгоритмы;
3. итераторы.

**Контейнеры (контейнерные классы)** – это способ организации хранения данных. Контейнеры в STL подключаются к программе с помощью шаблонных классов. Контейнеры бывают разных типов. Например: vector – определяет динамический массив, list – позволяет работать с линейным списком, map – обеспечивает доступ к значениям, имеющим уникальные ключи. В каждом классе-контейнере определяется набор функций для работы с этим контейнером.

**Алгоритмы** – это функции, которые производит действия над элементами контейнеров. Алгоритмы в STL не являются методами классов и не являются дружественными функциями по отношению к контейнерам. В современном стандарте алгоритмы – это независимые шаблонные функции, которые можно использовать при работе с обычными массивами С++. Существуют алгоритмы для инициализации, сортировки, поиска или замены содержимого контейнеров.

**Итераторы** – это объекты (в STL объекты класса iterator), которые по отношению к контейнерам играют роль указателей. Они позволяют получать доступ к содержимому контейнера так же, как указатели используются для доступа к элементам массива.

**Замечание**: Концепция STL в некотором смысле противоречит основным принципам ООП – происходит разделение данных и алгоритмов. Данные находятся под управлением контейнерных классов, а операции определяются адаптируемыми алгоритмами.

**Контейнеры, определенные STL**

STL включает в себя семь основных типов контейнеров и три производных типа.

Контейнеры STL подразделяются на *две категории*:

1. Последовательные.
2. Ассоциативные.

Последовательные

* векторы (динамические массивы);
* списки (идея связного списка) ;
* очереди с двусторонним доступом(деки) (комбинация стека и обычного списка. Стек работает по принципу LIFO, а очередь по принципу FIFO).

Ассоциативные

* множества;
* мультимножества;
* отображения;
* мультиотображения.

Производными от последовательных контейнеров являются еще три специализированных контейнера:

* стек;
* очередь;
* приоритетная очередь.

Специализированные контейнеры создаются из базовых с помощью конструкции, называемой *адаптером контейнера*. Эти контейнеры обладают более простым интерфейсом, чем обычные контейнеры.

Стек использует доступ к данным с одного конца. Очередь использует для “проталкивания ” данных один конец, а для “выталкивания” данных – другой. В приоритетной очереди данные проталкиваются в произвольном порядке, а выталкиваются в соответствии с величиной хранящегося значения. Приоритет имеют данные с наибольшим значением. Приоритетная очередь автоматически сортирует хранящуюся в ней информацию.

Для использования этих классов надо применить шаблон к шаблону.

**Пример**:

stack< deque<int> > aStak; /\* Объект типа стек, содержащий значения int, порожденный классом «очередь с двусторонним доступом» (deque). \*/

|  |  |  |
| --- | --- | --- |
| **Контейнер** | **Назначение** | **Заголовок** |
| **bitset** | Набор битов | **<bitset>** |
| **deque** | Двусторонняя очередь | **<deque>** |
| **list** | Линейный список | **<list>** |
| **map** | Хранит пары ключ-значение, в которых каждому ключу соответствует лишь одно значение | **<map>** |
| **multimap** | Хранит пары ключ-значение, в которых каждому ключу может соответствовать несколько значений | **<multimap>** |
| **multiset** | Множество, в которое элемент может входить несколько раз | **<multiset>** |
| **priority\_queue** | Очередь с приоритетами | **<queue>** |
| **queue** | Очередь | **<queue>** |
| **set** | Множество, в которое каждый элемент входит только один раз | **<set>** |
| **stack** | Стек | **<stack>** |
| **vector** | Динамический массив | **<vector>** |

Для работы с STL необходимо выбрать тип контейнера. Каждый из них обладает как преимуществами перед остальными, так и недостатками.

В последовательных контейнерах данные хранятся последовательно «в ряд».

Ассоциативные контейнеры

Ассоциативные контейнеры автоматически сортируют свои элементы (или ключи). По умолчанию сравнение идет при помощи оператора меньше (<). В ассоциативных контейнерах данные располагаются не последовательно, доступ к ним осуществляется посредством ключей (номеров строк, которые модифицируются контейнерами автоматически).

В STL есть два типа ассоциативных контейнеров: множества и отображения.

***Множества*** – коллекции, в которых элементы сортируются в соответствии с их значениями (каждый элемент уникален).

***Мультимножества*** – множества, которые могут содержать элементы с одинаковыми значениями.

***Отображения*** – коллекции, состоящие из пар “ ключ/значение”. У каждого элемента есть ключ, определяющий порядок сортировки. Отображение можно использовать как *ассоциативный массив*, то есть массив с произвольным типом индекса.

***Мультиотображения*** – отображения с возможностью дублирования ключей. То есть может быть несколько элементов с одинаковыми ключами.

Ассоциативные контейнеры обычно реализуются в виде *бинарных деревьев*, что позволяет производить быстро поиск, вставку или удаление данных. У каждого элемента есть один родитель и два потомка. Все предки слева от узла имеют меньшие значения, а предки справа – большие.

Множества в использовании проще и используются чаще, чем отображения.

Например, книги в библиотеке упорядочиваются по алфавиту, в качестве ключа используются фамилии автора.

**Алгоритмы**

Наиболее используемые алгоритмы STL

|  |  |
| --- | --- |
| **Алгоритм** | **Назначение** |
| **find** | возвращает первый элемент с указанным значением |
| **count** | считает количество элементов, имеющих указанное значение |
| **equal** | сравнивает содержимое двух контейнеров и возвращает true, если все соответствующие элементы эквивалентны |
| **search** | ищет последовательность значений в одном контейнере, которая соответствует такой же последовательности в другом |
| **copy** | копирует последовательность значений в другое место того же или другого контейнера |
| **swap** | обменивает значения, которые хранятся в разных местах |
| **iter\_swap** | обменивает последовательности значений, хранящиеся в разных местах |
| **fill** | копирует значение в последовательность ячеек |
| **sort** | сортирует значения в указанном порядке |
| **merge** | комбинирует два сортированных диапазона значений для получения набольшего диапазона (третьего) |
| **accumulate** | возвращает сумму элементов в заданном диапазоне |
| **for\_each** | выполняет указанную функцию для каждого элемента контейнера |

**Пример**: // алгоритмы – find(), count(), sort()

#include <iostream>

#include <algorithm>

using namespace std;

// Алгоритм find()

int arr[10] = { 5, 4, 7 , 8, 11, 3, 2, 1, 15, 7 };

int main()

{

setlocale(LC\_ALL, "Russian");

int\* ptr;

ptr = find(arr, arr + 10, 7);

cout << "Число 7 стоит на " << (ptr - arr);

cout << " месте массива." << endl;

// Алгоритм count()

int n = count(arr, arr + 10, 7);

cout << "Число 7 встречается " << n;

cout << " раз(а)." << endl;

// Алгоритм sort()

sort(arr, arr + 10);

for (int i = 0; i < 10; i++)

cout << arr[i] << ' ';

cout << endl;

system("pause");

return 0;

}

Результаты выполнения программы:

Число 7 стоит на 2 месте массива.

Число 7 встречается 2 раз(а).

1 2 3 4 5 7 7 8 11 15

Рассмотрим алгоритмы, работающие одновременно с несколькими контейнерами. Алгоритм merge() – работает с тремя контейнерами, объединяя элементы двух из них в третий, целевой контейнер. Для правильной работы первые два массива должны быть отсортированы.

**Пример**: // алгоритмы – search(), merge()

#include <iostream>

#include <algorithm>

using namespace std;

int source[] = { 1, 4, 3, 1, 2, 3, 1, 2, 4, 5, 1, 2, 3, 4, 5 };

int pattern[] = { 2, 3 };

int arr1[] = { 1, 3, 5, 7, 8, 9, 12 };

int arr2[] = { 2, 3, 6, 7, 8, 10 };

int arr3[13];

int main()

{

setlocale(LC\_ALL, "Russian");

// Алгоритм search()

int\* ptr;

ptr = search(source, source + 15, pattern, pattern + 2);

if (ptr == source + 15)

cout << "Совпадений не найдено" << endl;

else

cout << "Совпадение в позиции " << (ptr - source) << endl;

// Алгоритм merge()

merge(arr1, arr1 + 7, arr2, arr2 + 6, arr3);

for (int i = 0; i < 13; i++) // вывести arr3

cout << arr3[i] << ' ';

cout << endl;

system("pause");

return 0;

}

Результаты выполнения программы:

Совпадение в позиции 4

1 2 3 3 5 6 7 7 8 8 9 10 12

**Функциональные объекты и пользовательские функции**

Некоторым объектам в качестве параметра требуются так называемые *функциональные объекты*.

Функциональный объект – это объект шаблонного класса, в котором имеется единственный метод – перегружаемая операция (). Рассмотрим пример, в котором будет:

1. использован функциональный объект – greater<>() в качестве третьего аргумента алгоритма sort() и который изменит режим сортировки.
2. определена пользовательская функция для функционального объекта. Эта функция будет выполнять сравнение и ее адрес (имя) будет использовано вместо функционального объекта.

**Пример**: /\* 1. сортировка массива чисел по убыванию;

2. сортировка массива строк \*/

#include <iostream>

#include <algorithm> //для sort()

#include <functional> //для greater<>

#include <string> // для strcmp()

using namespace std;

int dat[8] = { 5, 8, -3, 4, 0, -1, 7, 4 };

char const \* names[] = { "Сергей", "Татьяна", "Елена", "Дмитрий", "Михаил", "Владимир" };

bool s\_comp(char const\*, char const\*); // объявление

int main()

{

setlocale(LC\_ALL, "Russian");

// Обратная сортировка чисел (функциональный объект greater<>())

sort(dat, dat + 8,greater<int>());

for (int i = 0; i < 8; i++) // вывести отсортированный массив

cout << dat[i] << ' ';

cout << endl;

// Сортировка строк (пользовательская функция)

sort(names, names + 6, s\_comp); // сортировка строк

// sort(names, names + 6); // сортировка строк

for (int j = 0; j < 6; j++) // вывод отсортированных строк

cout << names[j] << endl;

system("pause");

return 0;

}

bool s\_comp(char const\* s1, char const \* s2) // возвращает true

// если s1<s2

{

return (strcmp(s1, s2) < 0) ? true : false;

}

**Алгоритмы с \_if**

Некоторые алгоритмы имеют версии с окончанием \_if. Таким алгоритмам требуется третий аргумент, который называется *предикат* и является функциональным объектом или функцией.

Рассмотрим пример, в котором будет использован алгоритм ***count\_if()*** для подсчета числа вхождений фамилии, которая будет задаваться в пользовательской функции. Алгоритм ***count\_if()*** применяет пользовательскую функцию к каждому элементу выбранного диапазона.

**Пример**: // алгоритм – count\_if()

#include <iostream>

#include <algorithm>

#include <windows.h>

using namespace std;

//---------------------------------------------------------

bool isDon(char const \* name) // возвращает true, если

// name=="Иванов"

{

return name == "Иванов";

}

//---------------------------------------------------------

char const \* names[] = { "Петров", "Сидоров", "Елисеев", "Иванов", "Степанов", "Владимиров", "Иванов", "Кузнецов", "Иванов", "Петров", "Степанов", "Иванов", "Машин", "Андреев" };

int main()

{

setlocale(LC\_ALL, "Russian");

int n = count\_if(names, names + 14, isDon);

cout << "Всего в списке Ивановых " << n << endl;

return 0;

}