

Лекция.

Ассоциативные контейнеры

Содержание

Множества и мультимножества	4
Отображения (map) и мультиотображения (multimap).....	12
Ассоциативный массив	15
Необходимые методы	21
Порядок сортировки	21
Схожесть с базовыми типами	22
Поиск всех людей с указанным именем	26
Поиск всех людей с указанным номером телефона	26
Создание собственных функциональных объектов	32
Функциональные объекты и поведение контейнеров	37
Резюме	38

Введение

Мы убедились в том, что последовательные контейнеры (векторы, списки и очереди с двусторонним доступом) хранят данные в фиксированной линейной последовательности. Поиск конкретного элемента в таком контейнере (если неизвестен или недоступен его индекс или он не находится в одном из концов контейнера) — это длительная процедура, включающая в себя пошаговый переход от позиции к позиции.

В ассоциативных контейнерах элементы не организуются в последовательности. Они представляют собой более сложные структуры, что дает большой выигрыш в скорости поиска. Обычно ассоциативные контейнеры — это «деревья», хотя возможны и другие варианты, например таблицы. В любом случае, главным преимуществом этой категории контейнеров является скорость поиска.

Давайте поговорим о поиске. Он производится с помощью *ключей*, обычно представляющих собой одно численное или строковое значение, которое может быть как атрибутом объекта в контейнере, так и самим объектом.

Рассмотрим две основные категории ассоциативных контейнеров STL: множества и отображения.

Множество хранит объекты, содержащие ключи. Отображения можно представить себе как своего рода таблицу из двух столбцов, в первом из которых хранятся объекты, содержащие ключи, а во втором — объекты, содержащие значения.

Во множествах, как и в отображениях, может храниться только один экземпляр каждого ключа. А каждому ключу, в свою очередь, соответствует уникальное значение. Такой подход используется в словарях, когда каждому слову ставится в соответствие только одна статья. Но в STL есть способ обойти это ограничение. *Мультимножества* и *мультитоображения* аналогичны своим родственным контейнерам, но в них одному ключу может соответствовать несколько значений.

Ассоциативные контейнеры имеют несколько общих методов с другими контейнерами. Тем не менее, некоторые алгоритмы, такие, как `lower_bound()` и `equal_range()`, характерны только для них. Кроме того, некоторые методы могут *быть* применены для любых контейнеров, кроме ассоциативных. Среди них семейство методов проталкивания и выталкивания (`push_back()` и т. п.). Действительно нет особого смысла в их применении к данной категории контейнеров, поскольку все равно элементы должны проталкиваться и выталкиваться в определённом порядке, но не в конец или начало контейнера.

Как уже указывалось, ассоциативные контейнеры обеспечивают быстрый доступ к данным за счет того, что они, как правило, построены на основе сбалансированных деревьев поиска (стандартом регламентируется только интерфейс контейнеров, а не их реализация).

Существует пять типов ассоциативных контейнеров: словари (map), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset). Словари часто называют также ассоциативными массивами или отображениями.

Словарь построен на основе пар значений, первое из которых представляет собой ключ для идентификации элемента, а второе — собственно элемент. Можно сказать, что ключ ассоциирован с элементом, откуда и произошло название этих контейнеров. Например, в англо-русском словаре ключом является английское слово, а элементом — русское. Обычный массив тоже можно рассматривать как словарь, ключом в котором служит номер элемента. В словарях, описанных в STL, в качестве ключа может использоваться значение произвольного типа. Ассоциативные контейнеры описаны в заголовочных файлах <map> и <set>.

Множества и мультимножества

Множества <set> часто используются для хранения объектов пользовательских классов. На рис. 5 показана схема работы с этой категорией контейнеров. Объекты упорядочены, и целый объект является ключом.

Как и для словаря, элементы во множестве упорядочены. Повторяющиеся элементы во множество не заносятся.

Для работы с множествами в стандартной библиотеке определены алгоритмы.

Во множествах с дубликатами (multiset) ключи могут повторяться, поэтому операции вставки элемента всегда выполняется успешно, и функция insert возвращает итератор на вставленный элемент. Элементы с одинаковыми ключами хранятся во множестве в порядке их занесения. Функция find возвращает итератор на первый найденный элемент или end(), если ни одного элемента с заданным ключом не найдено.

При работе с одинаковыми ключами в multiset часто пользуются функциями count, lower_bound, upper_bound и equal_range, имеющими тот же смысл, что и для словарей с дубликатами.

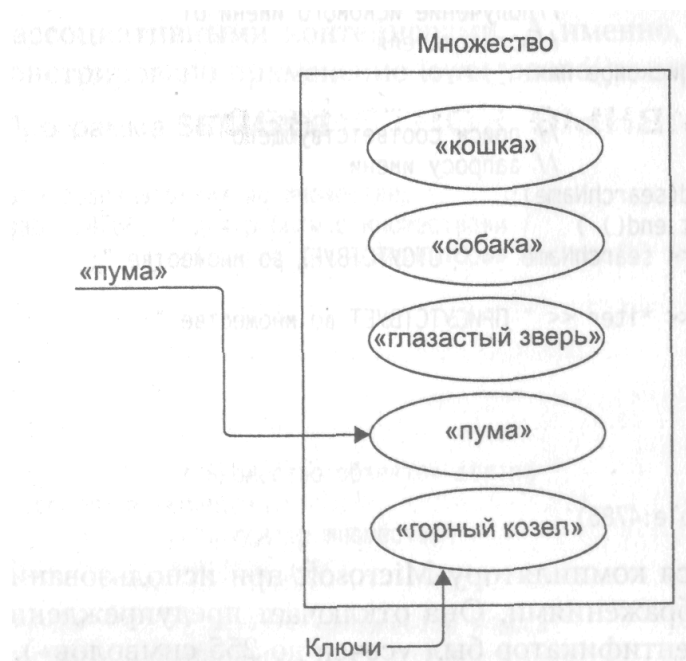


Рис. 5. Множество объектов типа string

В нашем первом примере SET мы продемонстрируем множество, содержащее объекты класса string.

Листинг 28. Программа SET

```
//-----

// set.cpp
// Множество, хранящее объекты типа string
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <set>
#include <string>
//-----
using namespace std;

template<class T>
void print(const T &container)//set<string>
{
    for (const auto &x : container)
    {
        cout << x << '\n';
    }
}
int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
```

```

    set<string> nameSet = { "Анастасия", "Роберт", "Мария", "Анна",
"Маргарита" };
    // Итератор для множества.
    set<string>::iterator iter;
    // Вставка элементов.
    nameSet.insert("Пётр");
    nameSet.insert("Яков");
    // Никакого эффекта: такой элемент уже имеется.
    nameSet.insert("Роберт");
    nameSet.insert("Борис");
    // Удаление элемента.
    nameSet.erase("Мария");
    // Вывод размера множества.
    cout << "\nSize=" << nameSet.size() << endl;
    // Вывод элементов множества.
    print(nameSet);
    //Получение искомого имени от пользователя
    string searchName;
    cout << "\nВведите имя для поиска: ";
    cin >> searchName;
    // Поиск соответствующего запросу имени.
    iter = nameSet.find(searchName);
    if (iter == nameSet.end())
        cout << "Имя " << searchName << " нет во множестве.";
    else
        cout << "Имя " << *iter << " во множестве.";
    cout << endl;
    system("pause");
    return 0;
}
//-----

```

Для определения множества необходимо указать тип хранимых в нем объектов. В нашем примере, например, это объекты класса string. К тому же необходимо указать функциональный объект, который будет использоваться для упорядочивания элементов множества. Здесь используется `less< string >`, по умолчанию.

Как видите, множество имеет интерфейс, во многом сходный с другими контейнерами STL. Мы можем инициализировать множество массивом, вставлять данные методом `insert()`, выводить их с помощью итераторов.

Для нахождения элементов мы используем метод `find()`. (Последовательные контейнеры имеют одноименный алгоритм.) Вот пример взаимодействия с программой SET. Здесь пользователь вводит в качестве искомого имени «Пётр»:

```

Size=7
Анастасия
Анна
Борис
Маргарита
Пётр
Роберт
Яков

Введите имя для поиска: Пётр
Имя Пётр во множестве.
Для продолжения нажмите любую клавишу . . .

```

Ниже приведён пример множества значений типа float.

```

//-----

// set.cpp
// Множество, хранящее объекты типа float
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <deque>
#include <functional>           //для greater<>
#include <set>
//-----
using namespace std;

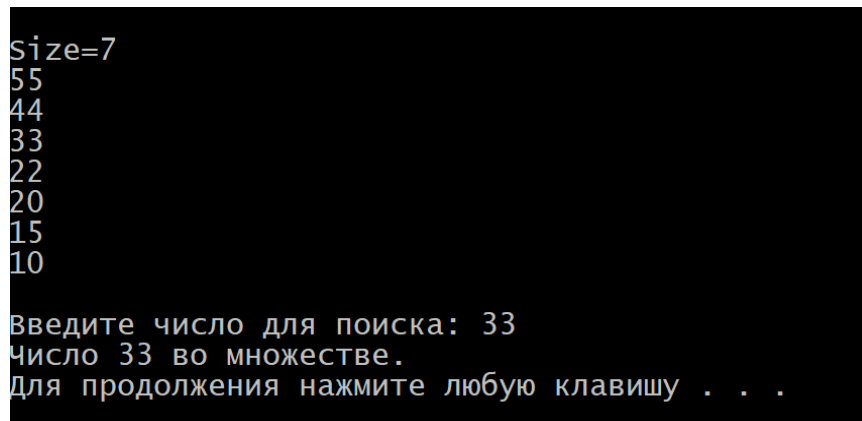
int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    // Массив объектов типа float
    float n[] = { 11, 22, 33, 44, 55 };
    // Инициализировать множество массивом.
    set<float, greater<float> > S(n, n + 5);
    // итератор для множества
    set<float, greater<float> >::iterator iter;
    // Дека объектов типа float
    deque<float> d = { 11, 22, 33, 44, 55 };
    // Инициализировать множество массивом.
    set<float, greater<float> > W(d.begin(), d.end());
    // вставка элементов
    S.insert(10);
    S.insert(15);
    // Никакого эффекта: такой элемент уже имеется.
    S.insert(10);
    S.insert(20);
    // Удаление элемента.
    S.erase(11);
    // Вывод размера множества.
    cout << "\nSize=" << S.size() << endl;
    // Вывод элементов множества.

```

```

    iter = S.begin();
    while( iter != S.end() )
        cout << *iter++ << '\n';
    //Получение искомого значения от пользователя.
    float search;
    cout << "\nВведите число для поиска: ";
    cin >> search;
    // Поиск соответствующего запросу значения.
    iter = S.find(search);
    if( iter == S.end() )
        cout << "Число " << search << " нет во множестве.";
    else
        cout << "Число " << *iter << " во множестве.";
    cout << endl;
    system("pause");
    return 0;
}
//-----

```



```

size=7
55
44
33
22
20
15
10

Введите число для поиска: 33
Число 33 во множестве.
Для продолжения нажмите любую клавишу . . .

```

Конечно, скорость поиска трудно заметить на примере таких крошечных множеств, однако, поверьте, при достаточно большом количестве элементов ассоциативные контейнеры сильно выигрывают в поиске по сравнению с последовательными контейнерами.

Рассмотрим одну довольно важную пару методов, которая может использоваться только с ассоциативными контейнерами. А именно, в нашем примере SETRANGE продемонстрировано применение `lower_bound()` и `upper_bound()`.

Листинг 29. Программа SETRANGE

```

//-----
// Листинг 3. Работа с диапазоном.cpp: определяет точку входа для
консольного приложения.
//
#include "stdafx.h"
#include <windows.h>

```



```

#include <iostream>
#include <set>
#include <string>
#include <functional>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    // Множество объектов string по умолчанию.
    set<string, greater<string> > countries =
    {
        "Австралия", "Бельгия", "Бразилия", "Великобритания",
        "Германия", "Греция", "Дания", "Зимбабве", "Ирландия"
    };
    // Итератор множества.
    set<string, greater<string> >::iterator iter;
    // Вставка компонентов класса organic.
    countries.insert("Албания");
    // Вывод множества.
    iter = countries.begin();
    while (iter != countries.end())
        cout << *iter++ << '\n';
    // Вывод значений из диапазона.
    string lower, upper;
    cout << "\nВведите диапазон (например, В А): ";
    cin >> lower >> upper;
    iter = countries.lower_bound(lower);
    while (iter != countries.upper_bound(upper))
        cout << *iter++ << '\n';
    system("pause");
    return 0;
}

//-----

```

Программа вначале выводит полный список элементов множества класса **countries**. Затем пользователя просят ввести пару ключевых значений, задающих диапазон внутри множества. Элементы, входящие в этот диапазон, выводятся на экран. Пример работы программы:

```
Ирландия
Зимбабве
Дания
Греция
Германия
Великобритания
Бразилия
Бельгия
Албания
Австралия

Введите диапазон (например, В А): В А
Бразилия
Бельгия
Албания
Австралия
Для продолжения нажмите любую клавишу . . .
```

Метод `lower_bound()` берет в качестве аргумента значение того же типа, что и ключ. Он возвращает итератор, указывающий на первую запись множества, значение которой не меньше аргумента (что значит «не меньше», в каждом конкретном случае определяется конкретным функциональным объектом, используемым при определении множества). Функция `upper_bound()` возвращает итератор, указывающий на элемент, значение которого не больше, чем аргумент. Обе эти функции позволяют, как мы и показали в нашем примере, задавать диапазон значений в контейнере.

В следующем примере множество по умолчанию упорядочивается в порядке возрастания ключей.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <set>
#include <string>
#include <functional>

using namespace std;

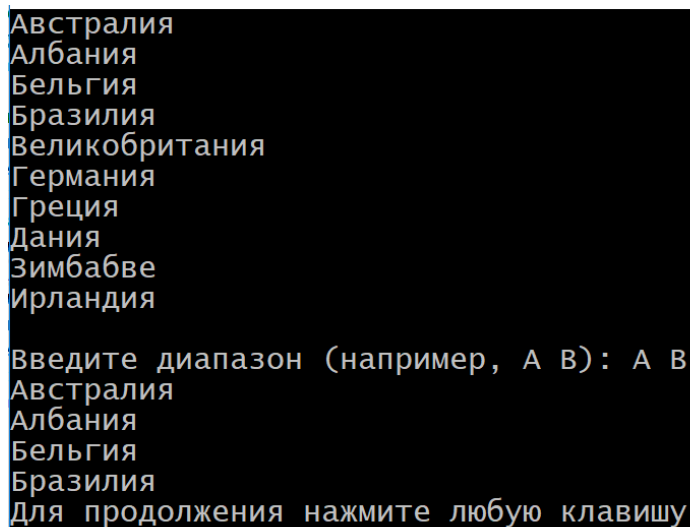
int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    // Множество объектов string по умолчанию - less<string>.
    set<string> countries =
    {
        "Австралия", "Бельгия", "Бразилия", "Великобритания",
        "Германия", "Греция", "Дания", "Зимбабве", "Ирландия"
    };
    // Итератор множества.
    set<string>::iterator iter;
    // Вставка компонентов класса organic.
    countries.insert("Албания");
```

```

// Вывод множества.
iter = countries.begin();
while (iter != countries.end())
    cout << *iter++ << '\n';
// Вывод значений из диапазона.
string lower, upper;
cout << "\nВведите диапазон (например, А Б): ";
cin >> lower >> upper;
iter = countries.lower_bound(lower);
while (iter != countries.upper_bound(upper))
    cout << *iter++ << '\n';
system("pause");
return 0;
}

```

Результат работы программы:



```

Австралия
Албания
Бельгия
Бразилия
Великобритания
Германия
Греция
Дания
Зимбабве
Ирландия

Введите диапазон (например, А Б): А В
Австралия
Албания
Бельгия
Бразилия
Для продолжения нажмите любую клавишу . . . _

```

Ниже приведён пример множества значений типа float.

```

//-----

// setrange.cpp
// Тестирование работы с диапазонами во множестве.
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <vector>
#include <set>
//-----
using namespace std;

//Шаблон функций Вывод элементов контейнера.
template<class T>

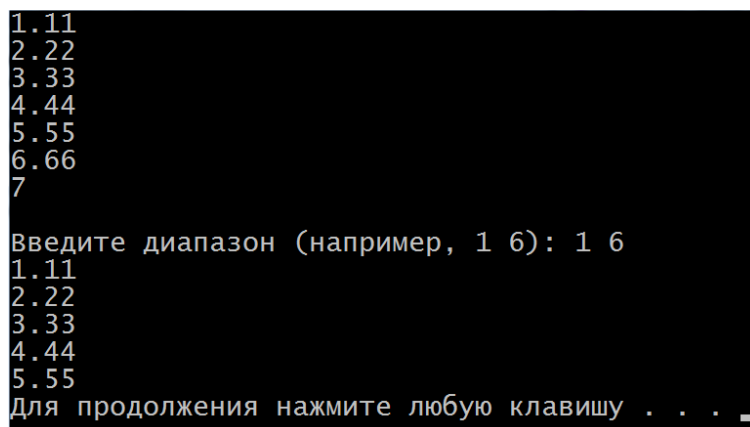
```

```

void print(const T &container)//set<string>
{
    for (const auto &x : container) cout << x << '\n';
}

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    vector<double> m = { 1.11, 2.22, 3.33, 4.44, 5.55, 6.66 };
    //Множество объектов float.
    set<double> S(m.begin(), m.end());
    //Итератор множества.
    set<double>::iterator iter;
    //Вставить элемент.
    S.insert(7);
    //Вывод множества.
    print(S);
    //-----Вывод значений из диапазона
    float lower, upper;
    cout << "\nВведите диапазон (например, 1 6): ";
    cin >> lower >> upper;
    iter = S.lower_bound(lower);
    while (iter != S.upper_bound(upper))
        cout << *iter++ << '\n';
    system("pause");
    return 0;
}
//-----

```



```

1.11
2.22
3.33
4.44
5.55
6.66
7
Введите диапазон (например, 1 6): 1 6
1.11
2.22
3.33
4.44
5.55
Для продолжения нажмите любую клавишу . . . _

```

Отображения (map) и мультиотображения (multimap)

В отображении (словаре, map), в отличие от словаря с дубликатами (мультиотображение, multimap), все ключи должны быть уникальны. Элементы в словаре хранятся в отсортированном виде, поэтому для ключей должно быть определено отношение «меньше».

В *отображении (словаре)* всегда хранятся пары значений, одно из

которых представляет собой ключевой объект, а другое — объект, содержащий значение. Ключевой объект содержит, естественно, ключ, по которому ищется значение. Объект-значение содержит некие данные, которые обычно и интересуют пользователя, запросившего что-то с помощью ключа. В ключевом объекте могут храниться строки, числа или более сложные объекты. Значениями зачастую являются тоже строки, числа или другие объекты, вплоть до контейнеров!

Например, ключом может быть слово, а значением — число, говорящее о том, сколько раз это слово встречалось в тексте. Такое отображение задает *частотную таблицу*. Или, например, ключом может быть слово, а значением — список номеров страниц. Такое решение может быть употреблено для создания индекса, как в конце книги. На рис. 6 показана ситуация, в которой ключами являются слова, а значениями — определения, как в обыкновенном словаре.

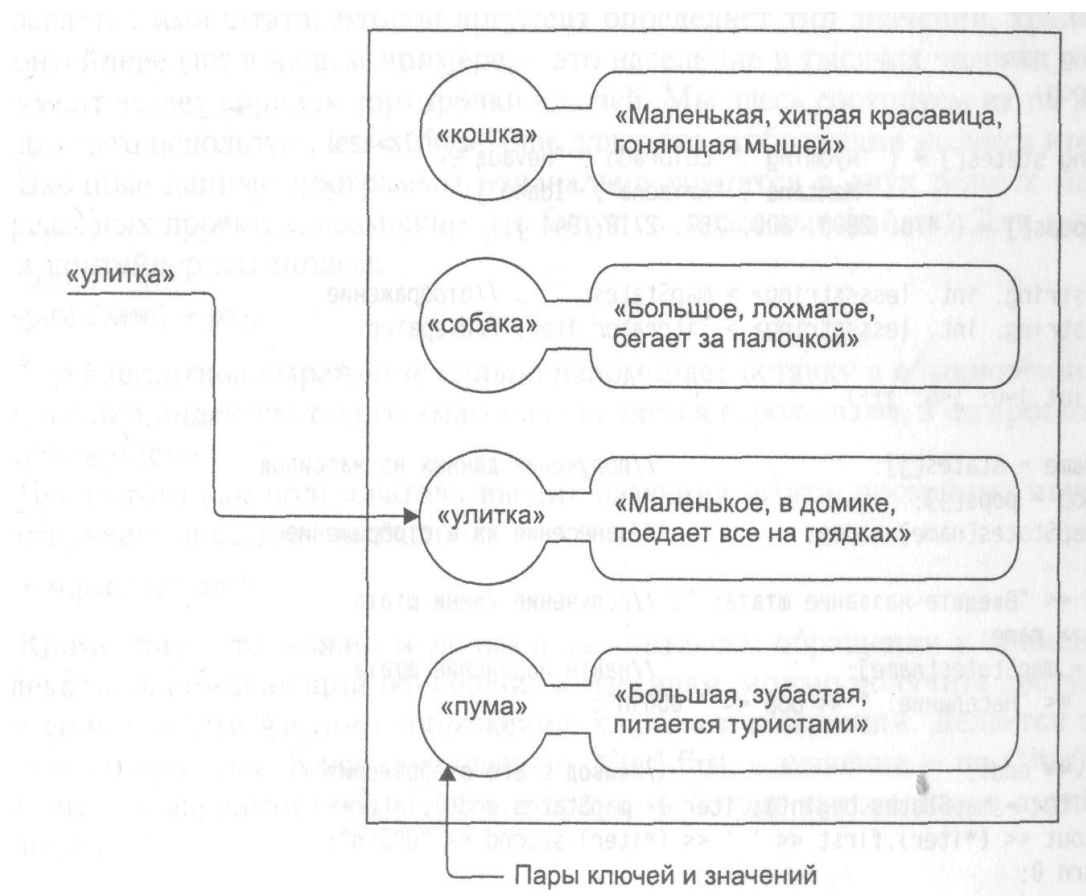


Рис. 6. Отображение «слово — фраза»

Как уже упоминалось, словари с дубликатами (multimap) допускают хранение элементов с одинаковыми ключами. Поэтому для них не определена операция доступа по индексу [], а добавление с помощью функции insert выполняется успешно в любом случае. Функция возвращает итератор на вставленный элемент.

Элементы с одинаковыми ключами хранятся в словаре в порядке их занесения. При удалении элемента по ключу функция `erase` возвращает количество удаленных элементов. Функция `equal_range` возвращает диапазон итераторов, определяющий все вхождения элемента с заданным ключом. Функция `count` может вернуть значение, большее 1. В остальном словари с дубликатами аналогичны обычным словарям.

Одним из самых популярных примеров использования отображений являются ассоциативные массивы. В обыкновенных массивах индекс всегда является целым числом. С его помощью, как известно, осуществляется доступ к конкретным элементам. Так, в выражении `anArray[3]` число 3 является индексом. Несколько иначе дело обстоит в ассоциативных массивах. Тип данных индекса массива можно задавать самостоятельно. При этом появляется уникальная возможность написать, например, такое выражение: `anArray[?jane?]`.

В качестве примера словаря приведём телефонную книгу, ключом в которой служит фамилия, а элементом — номер телефона:

```
#include "stdafx.h"
#include <windows.h>
#include <fstream>
#include <iostream>
#include <string>
#include <map>

using namespace std;
typedef map <string, long, less <string> > map_s1; // 1

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    map_s1 m1;
    ifstream in("phonebook.txt");
    string str;
    long num = 0;
    // Чтение номера.
    while ( in >> num, !in.eof())
    {
        in.get(); // Пропуск пробела
        getline(in, str); // Чтение фамилии
        m1[str] = num; // Занесение в словарь
        cout << str << " " << num << endl;
    }
    // Дополнение словаря.
    m1["Петя Р."] = 2134622;
    map_s1::iterator i ;
```

```

    cout << "m1:" << endl;
    // Вывод словаря.
    for (i = m1.begin(); i != m1.end(); i++)
        cout << (*i).first << " " << (*i).second << endl;
    i = m1.begin(); i++;
    cout << "Второй элемент: ";
    cout << (*i).first << " " << (*i).second << endl;
    // Вывод элемента по ключу.
    cout << "Коля: " << m1["Коля"] << endl;
    system("pause");
    return 0;
}

```

Для улучшения читаемости программы введено более короткое обозначение типа словаря (оператор, помеченный // 1). Сведения о каждом человеке расположены в файле phonebook на одной строке: сначала идет номер телефона, затем через пробел фамилия:

```

2718956 Вася
2663456 Катя П.
2115432 Николаев В.
3461727 Коля

```

Для итераторов словаря допустимы операции инкремента и декремента, но не операции + и -. Ниже приведен результат работы программы (обратите внимание, что словарь выводится в упорядоченном виде):

```

Вася 2718956
Катя П. 2663456
Николаев В. 2115432
Коля 3461727
m1:
Вася 2718956
Катя П. 2663456
Коля 3461727
Николаев В. 2115432
Петя Р. 2134622
Второй элемент: Катя П. 2663456
Коля: 3461727
Для продолжения нажмите любую клавишу . . .

```

Ассоциативный массив

Давайте рассмотрим небольшой пример отображения, используемого в качестве ассоциативного массива. Ключами будут названия государств, а значениями — их население.

Листинг 30. Программа ASSO_ARR

```

//-----
// Государства.cpp: определяет точку входа для консольного приложения.
//

```

```

#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <string>
#include <map>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    string name;
    int pop;

    string states[] = { "Россия", "Великобритания", "Франция",
        "Германия", "Италия", "Испания" };
    //Тысяч человек.
    int pops[] = { 146100, 63396, 64700, 80200, 59571, 47371 };
    //Отображение.
    map<string, int, less<string> > mapStates;
    //Итератор.
    map<string, int, less<string> >::iterator iter;

    for (int j = 0; j<6; j++)
    {
        //Получение данных из массивов.
        name = states[j];
        pop = pops[j];
        //Занесение их в отображение.
        mapStates[name] = pop;
    }
    //Получение имени страны.
    cout << "Введите наименование государства: ";
    cin >> name;
    //Найти население страны.
    pop = mapStates[name];
    cout.width(16);
    cout.left;
    cout << "Население: " << pop << " 000\n";

    cout << endl;
    //Вывод всего отображения.
    for (iter = mapStates.begin(); iter != mapStates.end(); iter++){
        cout.width(15);
        cout.left;
        cout << (*iter).first << ' ' << (*iter).second << "000\n";
    }

    system("pause");
    return 0;
}
//-----

```

При запуске программы у пользователя запрашивается название страны.

Полученное значение используется в качестве ключа для поиска в отображении значения населения и вывода его на экран. После этого выводится все содержимое отображения: все пары страна — население. Приведем пример работы программы:

```
Введите наименование государства: Италия
Население: 59571 000

Великобритания 63396000
Германия 80200000
Испания 47371000
Италия 59571000
Россия 146100000
Франция 64700000
Для продолжения нажмите любую клавишу . . .
```

При использовании множеств и отображений к скорости поиска никаких претензий не возникает. Вот и в данном случае программа очень быстро находит значение населения по введенному названию страны (а если в отображении хранятся миллионы пар, представляете, каким эффективным оказывается этот подход!). Ну, всегда приходится чем-то жертвовать, и итерация в данном типе контейнеров не такая быстрая, как в последовательных контейнерах. Обратите внимание, имена стран расположены в алфавитном порядке, хотя были заданы в массиве хаотическим образом.

Определение отображения имеет три шаблонных аргумента:

```
map<string, int, less<string> > maStates;
```

Первый из них задает тип ключа. В данном случае это `string`, потому что в строке задается имя страны. Второй аргумент определяет тип значений, хранящихся в контейнере (`int` в нашем примере — это население в тысячах человек). Третий аргумент задает порядок сортировки ключей. Мы здесь сортируем их по алфавиту, для чего используем `less<string>`. Его можно не указывать, он задан по умолчанию. Еще для этого отображения задается итератор.

Входные данные программы изначально хранятся в двух разных массивах (в реальных проектах, возможно, это будут специальные файлы). Для занесения их в контейнер мы пишем:

```
mapStates[name] = pop;
```

Это элегантное выражение сильно напоминает вставку в обыкновенный массив, однако индексом такого «массива» является строка `name`, а не просто какое-то целое число.

После того как пользователь вводит название страны, программа ищет

соответствующее значение населения:

```
pop = mapStates[name];
```

Кроме того, что можно использовать синтаксис обращения к элементам по индексам, напоминающий обращение к массивам, можно получить доступ одновременно к обеим частям отображения: ключам и значениям. Делается это при помощи итераторов. Ключ получают по `(*iter).first`, а значение — по `(*iter).second`. При других вариантах обращения итератор работает так же, как в других контейнерах.

Хранение пользовательских объектов

До этого момента в наших примерах мы хранили объекты базовых типов. Между тем, одно из достоинств STL заключается как раз в том, что объекты пользовательских типов тоже являются полноправными данными, которые можно хранить, над которыми можно выполнять все те же операции. В этом разделе мы рассмотрим примеры работы с типами пользователя.

Множество объектов `person`

Начнем наш разговор с класса `person`, в котором содержатся фамилии, имена и телефоны людей. Мы будем создавать компоненты этого класса, и вставлять их во множество, заполняя тем самым виртуальную телефонную книгу. Пользователь взаимодействует с программой, вводя имя человека. На экран выводится результат поиска данных, соответствующих указанному имени. Применим для решения этой задачи мультимножество, чтобы два и более объекта `person` могли иметь одинаковые имена. Приведем листинг программы SETPERS.

Листинг 31. Программа SETPERS

```
// Листинг 6. Person_multiset.cpp: определяет точку входа для
//консольного приложения.
//

#include "stdafx.h"
#include <windows.h>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <set>

using namespace std;
//Шаблон функций Вывод элементов контейнера.
template<class T>
void print(const T &container)
```

```

{
    for (const auto &x : container) cout << x.ToString() << '\n';
}

class person
{
private:
    string lastName;
    string firstName;
    long phoneNumber;
public:
    // Конструктор по умолчанию.
    person(): lastName("пусто"), firstName("пусто"), phoneNumber(0)
    { }
    // Конструктор с тремя параметрами.
    person(string lana, string fina, long pho):
        lastName(lana), firstName(fina), phoneNumber(pho)
    { }
    //Оператор < для класса person.
    bool operator < (const person& p) const
    {
        if (lastName == p.lastName)
            return (firstName < p.firstName) ? true : false;
        return (lastName < p.lastName) ? true : false;
    }
    //Оператор == для класса person.
    bool operator==(const person& p) const
    {
        return (lastName == p.lastName &&
            firstName == p.firstName) ? true : false;
    }
    //Преобразование в формат строки
    string ToString(void) const
    {
        ostringstream os;
        os.width(10);
        os.left;
        os << lastName;
        os.width(10);
        os.right;
        os << firstName;
        os.width(10);
        os << "Т-ф: " << phoneNumber;
        return os.str();
    }
};

int _tmain(int argc, _TCHAR* argv[])
{

```

```

SetConsoleCP(1251);
SetConsoleOutputCP(1251);
//Мультимножество класса person.
multiset< person > persSet =
{
    { "Иванов", "Фёдор", 8435150 },
    { "Петров", "Борис", 3327563 },
    { "Фёдоров", "Пётр", 6946473 },
    { "Мартынов", "Валерий", 4157300 },
    { "Кузин", "Евгений", 9207404 },
    { "Сидоров", "Николай", 8435150 },
    { "Кошкин", "Семён", 7049982 },
    { "Фёдоров", "Пётр", 7764987 }

};
//Итератор этого мультимножества.
multiset<person>::iterator iter;
//Занести объекты person в мультимножество.
persSet.insert(person("Фёдоров", "Пётр", 1264987));

cout << "\nКоличество записей: " << persSet.size() << endl;

//Вывод содержимого мультимножества.
print(persSet);
//Получение имени и фамилии.
string searchLastName, searchFirstName;
cout << "\n\nДля поиска " << endl;
cout << "Введите фамилию: ";
cin >> searchLastName;
cout << "Введите имя: ";
cin >> searchFirstName;
//-----Создание объекта с заданными значениями атрибутов.
person searchPerson(searchLastName, searchFirstName, 0);
//-----Сосчитать количество людей с таким именем.
int cntPersons = persSet.count(searchPerson);
cout << "Число людей с таким именем: " << cntPersons << endl;
//-----Вывести все записи, отвечающие запросу.
iter = persSet.lower_bound(searchPerson);
while( iter != persSet.upper_bound(searchPerson) )
    cout << (*iter++).ToString() << endl;
cout << endl;
system("pause");
return 0;
}

```

```

Количество записей: 9
Иванов    Фёдор    т-ф: 8435150
Кошкин    Семён    т-ф: 7049982
Кузин     Евгений  т-ф: 9207404
Мартынов  Валерий  т-ф: 4157300
Петров    Борис    т-ф: 3327563
Сидоров   Николай  т-ф: 8435150
Фёдоров   Пётр     т-ф: 6946473
Фёдоров   Пётр     т-ф: 7764987
Фёдоров   Пётр     т-ф: 1264987

```

```

Для поиска
Введите фамилию: Фёдоров
Введите имя: Пётр
Число людей с таким именем: 3
Фёдоров    Пётр    т-ф: 6946473
Фёдоров    Пётр    т-ф: 7764987
Фёдоров    Пётр    т-ф: 1264987

```

Необходимые методы

Для работы с контейнерами STL классу `person` требуется несколько общих методов. Они представляют собой конструкторы по умолчанию (без аргументов), которые в данном примере не очень нужны, но вообще-то обычно бывают довольно полезны. Кроме того, весьма и весьма полезными оказываются перегружаемые операции `<` и `==`. Все эти методы используются списковым классом и различными алгоритмами. В других ситуациях могут понадобиться какие-то другие специфические методы. (Как и при работе с большинством классов, наверное, нелишним будет иметь под рукой перегружаемую операцию присваивания, конструктор копирования и деструктор, но сейчас мы не будем заводить разговор об этом, дабы не загромождать листинг.)

Перегружаемые операции `<` и `==` должны иметь константные параметры. Вообще говоря, даже лучше сделать их дружественными, но и в виде обычных методов они вполне подходят для работы.

Порядок сортировки

Перегружаемая операция `<` задает метод сортировки элементов множества. В нашей программе `SETPERS` мы определяем его таким образом, чтобы сортировались фамилии, а в случае их совпадения — еще и имена.

Ниже приводится некий пример взаимодействия пользователя с программой. Вначале на экран выводится весь список хранимых значений (конечно, так не следует делать в случае, если вы имеете дело с реальной базой данных с большим числом элементов). Поскольку данные хранятся в мультимножестве, они сортируются автоматически. После этого у пользователя запрашивается имя человека, он вводит «Фёдоров», затем

«Пётр» (вначале вводится фамилия). А в нашем списке имеется три полных тезки, и на экран выводится информация о троих.

```
Количество записей: 9
Иванов      Фёдор      т-ф: 8435150
Кошкин      Семён      т-ф: 7049982
Кузин       Евгений    т-ф: 9207404
Мартинов    Валерий    т-ф: 4157300
Петров      Борис      т-ф: 3327563
Сидоров     Николай    т-ф: 8435150
Фёдоров     Пётр       т-ф: 6946473
Фёдоров     Пётр       т-ф: 7764987
Фёдоров     Пётр       т-ф: 1264987

Для поиска
Введите фамилию: Фёдоров
Введите имя: Пётр
Число людей с таким именем: 3
Фёдоров     Пётр       т-ф: 6946473
Фёдоров     Пётр       т-ф: 7764987
Фёдоров     Пётр       т-ф: 1264987
```

Схожесть с базовыми типами

Как видите, едва мы определили класс, как получили возможность работы контейнера с ним теми же способами, что и с обычными переменными базовых типов.

Метод `size()` используется для вывода количества записей. Затем производится итерация по контейнеру с целью вывода на экран всех элементов.

Поскольку мы имеем дело с мультимножеством, методы `lower_bound()` и `upper_bound()` дают возможность выводить все элементы, входящие в указанный диапазон. В нашем примере верхняя граница совпадает с нижней, поэтому просто выводятся все записи, содержащие данные о людях с заданным именем. Обратите внимание: мы создаем фиктивную запись с данными о человеке (или о нескольких людях) с тем же именем (именами), которое пользователь указал в запросе. При этом значения функций `lower_bound()` и `upper_bound()` должны соответствовать именно указанным записям, в отличие от ситуации, описанной выше (в самом списке их значения совпадали).

Список объектов класса `person`

Поиск человека с заданным именем оказывается очень быстрым при работе с множествами или мультимножествами. Если же нам важнее не быстрый поиск, а быстрая вставка или удаление объектов типа `person`, то лучше обратиться к спискам. В следующем примере показано, как именно

применять списки объектов пользовательских классов на практике.

Листинг 32. Программа LISTPERS

```
// Список Персон.cpp: определяет точку входа для консольного
//приложения.
//
// Использование списка для хранения объектов person
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <sstream>
#include <list>
#include <algorithm>
#include <string>
using namespace std;
//-----

class person
{
private:
    string lastName;
    string firstName;
    long phoneNumber;
public:
    //-----конструктор без аргументов
    person() :
        lastName("space"), firstName("space"), phoneNumber(0L)
    { }
    //-----конструктор с тремя аргументами
    person(string lana, string fina, long pho) :
        lastName(lana), firstName(fina), phoneNumber(pho)
    { }
    //-----перегруженные операторы сравнения
    friend bool operator<(const person&, const person&);
    friend bool operator==(const person&, const person&);
    friend bool operator!=(const person&, const person&);
    friend bool operator>(const person&, const person&);
    //-----преобразование в формат строки
    string ToString(void) const
    {
        ostringstream os;
        os.width(10);
        os.left;
        os << lastName;
        os.width(10);
        os.right;
        os << firstName;
        os.width(10);
        os << "телефон: " << phoneNumber;
        return os.str();
    }
};
```

```

    }
    //-----вернуть номер телефона
    long get_phone() const
    {
        return phoneNumber;
    }
};
//-----перегруженный == для класса person
bool operator==(const person& p1, const person& p2)
{
    return (p1.lastName == p2.lastName &&
        p1.firstName == p2.firstName) ? true : false;
}
//-----перегруженный < для класса person
bool operator<(const person& p1, const person& p2)
{
    if (p1.lastName == p2.lastName)
        return (p1.firstName < p2.firstName) ? true : false;
    return (p1.lastName < p2.lastName) ? true : false;
}
//-----перегруженный != для класса person
bool operator!=(const person& p1, const person& p2)
{
    return !(p1 == p2);
}
//-----перегруженный > для класса person
bool operator>(const person& p1, const person& p2)
{
    return !(p1<p2) && !(p1 == p2);
}
//-----
int _tmain(int argc, _TCHAR* argv[])
{
    //-----список объектов типа person
    list<person> persList;
    //-----итератор для этого списка
    list<person>::iterator iter1;
    //-----занести объекты в список
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    //-----Создание объектов person и занесение объектов в список
    persList.push_back(person("Иванов", "Фёдор", 8435150));
    persList.push_back(person("Петров", "Борис", 3327563));
    persList.push_back(person("Фёдоров", "Пётр", 6946473));
    persList.push_back(person("Мартынов", "Валерий", 4157300));
    persList.push_back(person("Кузин", "Евгений", 9207404));
    persList.push_back(person("Сидоров", "Николай", 8435150));
    persList.push_back(person("Кошкин", "Семён", 7049982));
    persList.push_back(person("Фёдоров", "Пётр", 7764987));

    cout << "Количество записей: " << persList.size() << endl;
}

```



```

//-----вывод содержимого списка
    iter1 = persList.begin();
    while (iter1 != persList.end())
        cout << (*iter1++).ToString() << endl;
//-----находим информацию о людях с именами
//-----и фамилиями, указанными в запросе

    string searchLastName, searchFirstName;
    cout << "\nВведите фамилию для поиска: ";
    cin >> searchLastName;
    cout << "Введите имя: ";
    cin >> searchFirstName;
//-----создаём персону с таким именем
    person searchPerson(searchLastName, searchFirstName, 0L);
//-----поиск по списку первого вхождения искомого значения
    iter1 = find(persList.begin(), persList.end(), searchPerson);
//-----поиск следующих совпадений
    if (iter1 != persList.end())
    {
        cout << "Этот человек в списке. " << endl;
        do
        {
            //вывод найденных записей
            cout << (*iter1).ToString() << endl;
            ++iter1; //продолжение поиска со следующей записи
            iter1 = find(iter1, persList.end(), searchPerson);
        } while (iter1 != persList.end());
    }
    else
        cout << "Этого человека нет в списке." << endl;
//-----находим человека по номеру телефона

    cout << "\nВведите номер телефона (1234567): ";
    long sNumber;
//-----получить искомый номер
    cin >> sNumber;
//-----итерация по списку
    bool found_one = false;
    for (iter1 = persList.begin(); iter1 != persList.end(); ++iter1)
    {
        if (sNumber == (*iter1).get_phone()) //сравнить номера
        {
            if (!found_one)
            {
                cout << "Этот человек в списке. " << endl;
                found_one = true;
            } //display the match
            cout << (*iter1).ToString() << endl;
        }
    } //end for
    if (!found_one)
        cout << "Этого человека нет в списке.";

```

```
    cout << endl;
    system("pause");
    return 0;
}
```

//-----

Поиск всех людей с указанным именем

В такой ситуации у нас нет возможности использовать методы `lower_bound()` и `upper_bound()`, поскольку в данном случае мы имеем дело и не с множеством, и не с отображением. В списках такие методы отсутствуют. Вместо них мы пользуемся уже знакомой функцией `find()`. Если функция возвращает значение найденного элемента, нужно снова вызвать ее для осуществления поиска, начиная со следующей записи после найденной. Это несколько усложняет программу, так как приходится вносить в нее цикл с двумя вызовами `find()`.

Поиск всех людей с указанным номером телефона

Сложнее обстоит дело с поиском данных при заданном номере телефона, так как методы этого класса, в том числе `find()`, предназначены для поиска первичной характеристики. Поэтому приходится «вручную» искать номер телефона, производя итерацию по списку и сравнивая заданный номер телефона с имеющимся в очередной текущей записи:

```
if( sNumber == (*iter1).getphone() )
```

Для начала программа просто выводит все записи из списка. Затем запрашивает у пользователя имя и находит соответствующие данные о человеке (или нескольких людях). После этого пользователя просят ввести номер телефона, и программа снова ищет подходящие данные.

Один из примеров работы программы представленной в примере:

```

Количество записей: 8
  Иванов    Фёдор телефон: 8435150
  Петров    Борис телефон: 3327563
  Фёдоров   Пётр телефон: 6946473
  Мартынов  Валерий телефон: 4157300
  Кузин     Евгений телефон: 9207404
  Сидоров   Николай телефон: 8435150
  Кошкин    Семён телефон: 7049982
  Фёдоров   Пётр телефон: 7764987

Введите фамилию для поиска: Фёдоров
Введите имя: Пётр
Этот человек в списке.
  Фёдоров    Пётр телефон: 6946473
  Фёдоров    Пётр телефон: 7764987

Введите номер телефона (1234567): 843515
Этот человек в списке.
  Иванов    Фёдор телефон: 8435150
  Сидоров   Николай телефон: 8435150

Для продолжения нажмите любую клавишу .

```

Как видите, программа нашла двух человек по указанному имени и двух человек по указанному телефону.

При использовании списков для хранения объектов, нужно объявлять четыре оператора сравнения для конкретного класса: `==`, `!=`, `<` и `>`. В зависимости от того, какие алгоритмы используются в программе, определяют те или иные из них, ведь далеко не всегда нужны все операторы сразу. Так, в нашем примере мы могли бы ограничиться лишь определением оператора `==`, хотя для полноты картины определили все. А если бы, например, в программе встречался алгоритм `sort()` для списка, была бы необходима перегружаемая операция `<`.

Функциональные объекты

Это понятие широко используется в STL. Одно из частых применений функциональных объектов является передача их в качестве аргументов алгоритмам. Таким образом, можно управлять поведением алгоритмов. Мы уже упоминали о функциональных объектах и даже использовали один из них в программе `SORTTEMP`. Там мы показывали пример предопределенного функционального объекта `greater<>()`, используемого для сортировки данных в обратном порядке. В этом параграфе будут представлены другие предопределенные функциональные объекты; кроме того, вы узнаете, как написать свой, который, возможно, предоставит даже больший контроль над алгоритмами STL.

Давайте вспомним, что такое функциональный объект. Это просто-

напросто функция, которая таким образом пристраивается к классу, что выглядит совсем как обычный объект. Тем не менее, в таком классе не может быть компонентных данных, а есть только один метод: перегружаемая операция (). Класс часто делают шаблонным, чтобы можно было работать с разными типами данных.

Предопределенные функциональные объекты

Предопределенные функциональные объекты расположены в заголовочном файле `FUNCTIONAL` и показаны в табл. 10. Там находятся объекты, которые могут работать с большинством операторов C++. В таблице символом `T` обозначен любой класс: пользовательский или одного из базовых типов. Переменные `x` и `y` — объекты класса `T`, передаваемые в функцию в виде параметров.

Таблица 10. Предопределенные функциональные объекты

Функциональный объект	Возвращаемое значение
<code>T = plus(T, T)</code>	<code>x+y</code>
<code>T = minus(T, T)</code>	<code>x-y</code>
<code>T = times(T, T)</code>	<code>x*y</code>
<code>T = divide(T, T)</code>	<code>x/y</code>
<code>T = modulus(T, T)</code>	<code>x%y</code>
<code>T = negate(T)</code>	<code>-x</code>
<code>bool = equal_to(T, T)</code>	<code>x==y</code>
<code>bool = not_equal_to(T, T)</code>	<code>x != y</code>
<code>bool = greater(T, T)</code>	<code>x > y</code>
<code>bool = less(T, T)</code>	<code>x < y</code>
<code>bool = greater_equal(T, T)</code>	<code>x >= y</code>
<code>bool = less_equal(T, T)</code>	<code>x <= y</code>
<code>bool = logical_and(T, T)</code>	<code>x&&y</code>
<code>bool = logical_or(T, T)</code>	<code>x y</code>
<code>bool = logical_not(T, T)</code>	<code>!x</code>

Из таблицы видно, что определены функциональные объекты для выполнения арифметических, логических операций и операций сравнения. Рассмотрим пример, в котором используется арифметический функциональный объект. Будем работать с классом `Frac`, представляющим простую дробь. В программе демонстрируется функциональный объект `plus<>()`, с помощью которого можно работать со значениями объектов `Frac` в контейнере. Приведем ее листинг.

Листинг 33. Программа PLUSFRAC

```

// Функция_объект_Плюс.cpp: определяет точку входа для консольного
приложения.
//

#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <sstream>
#include <list>
#include <algorithm>
#include <numeric>           //для accumulate()
#include <string>
//-----
using namespace std;

//-----
class Frac
{
private:
    int num;           //Числитель
    int denom;         //Знаменатель
public:
    //конструктор по умолчанию
    Frac(): num(0), denom(1)
    { }
    //конструктор с двумя аргументами
    Frac(int n, int d) : num(n), denom(d)
    { }
    void display() const //вывод на экран
    {
        cout << num << '/' << denom << endl;
    }
    //ввод данных пользователем
    void get()
    {
        char dummy;
        cout << "\nДробь (формат -1/2): ";
        cin >> num >> dummy >> denom;
    }
    //перегружаемая операция +
    Frac operator + (const Frac right) const
    {
        //добавить компоненты
        int tempnum = num*right.denom + denom*right.num;
        int tempdenom = denom* right.denom;
        if (tempdenom < 0) //приведение знака к числителю
        {
            tempnum *= -1;
            tempdenom *= -1;
        }
        return Frac(tempnum, tempdenom); //возврат суммы
    }
}

```

```

//перегружаемая операция +
Frac operator - () const
{
    return Frac(num, -denom); //меняем знак
}
//перегруженный оператор ==
bool operator == (const Frac& right) const
{
    return (num*right.denom + denom*right.num - denom *
right.denom == 0 ? true : false);
}
//перегруженный оператор <
bool operator < (const Frac& right) const
{
    return ((*this+(-right)).num < 0);
}
//перегружаемая операция !=
bool operator != (const Frac& at2) const
{
    return !(*this == at2);
}
//перегружаемая операция >
bool operator >(const Frac& at2) const
{
    return !(*this<at2) && !(*this == at2);
}
}; //конец класса Frac
//-----
int _tmain(int argc, _TCHAR* argv[])
{
    char answer;
    Frac temp, sum;
    list<Frac> Fraclist; //список типа Fraclist
    list<Frac>:: iterator i; //список типа Fraclist

    //-----кирилизация ввода/вывода
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    do { //получить значения от пользователя
        temp.get();
        Fraclist.push_back(temp);
        cout << "Продолжить (д/н)? ";
        cin >> answer;
    } while (answer != 'н');
    //вывод на экран
    foreach (temp in Fraclist)
    {
        temp.display();
    }
    for (i = Fraclist.begin(); i != Fraclist.end(); i++)
        (*i).display();
}

```

```

//суммировать все элементы
sum = accumulate(Fraclist.begin(), Fraclist.end(),
    Frac(0, 1), plus<Frac>());
cout << "\nСумма = ";
sum.display();           //вывод суммы на экран
cout << endl;
system("pause");
return 0;
}

```

Кроме функционального объекта, в этой программе представляется алгоритм `accumulate()`. Существуют две версии этой функции. Версия с тремя аргументами всегда суммирует (с помощью перегруженного `+`) значения из заданного диапазона. Если же у алгоритма четыре аргумента, как в нашем примере, то может быть использован любой функциональный объект из показанных в табл. 10.

Четыре аргумента `accumulate()` — это итераторы первого и последнего элемента диапазона, инициализационное значение суммы (обычно 0), а также операция, которую следует применить к элементам. В нашей программе мы выполняем сложение (`plus<>()`), но можно было бы выполнять и вычитание, и деление, и умножение, и другие операции, задаваемые функциональными объектами. Вот пример взаимодействия с программой:

```

Дробь (формат -1/2): -1/2
Продолжить (д/н)? д

Дробь (формат -1/2): 3/4
Продолжить (д/н)? д

Дробь (формат -1/2): 5/6
Продолжить (д/н)? д

Дробь (формат -1/2): -4/5
Продолжить (д/н)? н
-1/2
3/4
5/6
-4/5
-1/2
3/4
5/6
-4/5

Сумма = 68/240

Для продолжения нажмите любую клавишу . . .

```

Алгоритм `accumulate()` не только яснее и понятнее, чем итерация по контейнеру, но и его использование дает больший эффект.

Функциональный объект `plus<>()` нуждается в перегруженном операторе `+`. Причем он должен быть перегружен для класса `Frac`. Этот оператор должен быть объявлен как `const-функция`, так как именно это требуется функциональному объекту `plus<>()`.

Прочие арифметические функциональные объекты работают аналогично. Логические функциональные объекты, такие, как `logical_and<>()`, следует использовать с объектами тех классов, где такое булево представление значения имеет смысл. Например, в качестве объекта может выступать обычная переменная типа `bool`.

Создание собственных функциональных объектов

Если ни один из готовых predefined функциональных объектов вас не устраивает, можно без особых сложностей написать свой. В нашем следующем примере освещаются две ситуации, в которых именно этот способ оказывается предпочтительным. Одна из них включает в себя применение алгоритма `sort()`, а другая — алгоритма `for_each()`.

Группу элементов очень удобно сортировать, используя отношения, заданные оператором `<` класса. Но задумывались ли вы когда-нибудь о том, что может произойти, если задаться целью отсортировать контейнер, содержащий не сами объекты, а только ссылки, указатели на них? Хранение указателей вместо объектов — это грамотное решение, особенно в случае больших объемов информации. Такой подход становится эффективным за счет того, что позволяет избежать копирования каждого объекта при помещении его в контейнер. Тем не менее, проблема сортировки остается, поскольку объекты будут выстроены по адресам указателей, а не по каким-то собственным их атрибутам.

Чтобы заставить `sort()` работать в случае с контейнером указателей так, как нам нужно, необходимо иметь отдельный функциональный объект, задающий метод сортировки.

Программа, листинг которой представлен чуть ниже, начинается с того, что задается вектор указателей на объекты класса `person`. Объекты помещаются в вектор, затем сортируются обычным способом, что, разумеется, приводит к упорядочиванию по адресам указателей, а не по атрибутам самих объектов. Это совсем не то, что нам нужно; кроме того, наша сортировка вообще не вызывает никакого эффекта, так как данные изначально вводились подряд, а значит, адреса указателей тоже следовали в возрастающем порядке. После этого вектор сортируется с учетом возникшей непонятной ситуации. Для этого мы используем собственный функциональный объект `comparePersons()`. Он упорядочивает элементы, на которые ссылаются указатели, а не сами значения указателей. В результате объекты `person` упорядочиваются в алфавитном порядке. Мы за это и

боролись. Приведем листинг.

Листинг 34. Программа SORTPTRS

```
// Функция_объект_для_Персоны.cpp: определяет точку входа для
//консольного приложения.
//
//-----
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <sstream>
#include <vector>
#include <algorithm>
#include <string>
//-----

using namespace std;
//-----
class person
{
private:
    string lastName;
    string firstName;
    long phoneNumber;
public:
    //--конструктор по умолчанию
    person() :
        lastName("space"), firstName("space"), phoneNumber(0L)
    { }
    //--конструктор с тремя аргументами
    person(string lana, string fina, long pho) :
        lastName(lana), firstName(fina), phoneNumber(pho)
    { }
    //friend bool operator<(const person&, const person&);
    //friend bool operator==(const person&, const person&);
    //-----преобразование в формат строки
    string ToString(void) const
    {
        ostringstream os;
        os.width(10);
        os.left;
        os << lastName;
        os.width(10);
        //os.right;
        os << firstName;
        os.width(10);
        os << "Т-ф : " << phoneNumber;
        return os.str();
    }
}
//-----вывод персональных данных
```

```

    void display() const
    {
        cout << endl << ToString();
        //cout << endl << lastName << ",\t" << firstName <<
        "\t\tphoneNumber: " << phoneNumber;
    }
//-----перегруженный < для класса person
    bool operator < (const person& p2) const
    {
        if (lastName == p2.lastName)
            return (firstName < p2.firstName) ? true : false;
        return (lastName < p2.lastName) ? true : false;
    }
//-----перегруженный == для класса person
    bool operator == (const person& p2) const
    {
        return (lastName == p2.lastName &&
            firstName == p2.firstName) ? true : false;
    }
    long get_phone() const // вернуть телефонный номер
    {
        return phoneNumber;
    }
}; //end class person
//-----
// функциональный объект для сравнения содержимого по указателям на
//объекты person
class comparePersons
{
public:
    bool operator() (const person* ptrP1,
        const person* ptrP2) const
    {
        return *ptrP1 < *ptrP2;
    }
};
//-----
//функциональный объект для вывода персональных данных, хранимых по
//указателям
class displayPerson
{
public:
    void operator() (const person* ptrP) const
    {
        ptrP->display();
    }
};
int _tmain(int argc, _TCHAR* argv[])
{
//-----кирилизация ввода/вывода

```

```

        SetConsoleCP(1251);
        SetConsoleOutputCP(1251);
//-----вектор указателей на объекты person
        vector<person*> vectPtrsPers;
//-----создание персональных данных
        person* ptrP1 = new person("Иванов", "Фёдор", 8435150);
        person* ptrP2 = new person("Петров", "Борис", 3327563);
        person* ptrP3 = new person("Фёдоров", "Пётр", 6946473);
        person* ptrP4 = new person("Мартынов", "Валерий", 4157300);
        person* ptrP5 = new person("Кузин", "Евгений", 9207404);
        person* ptrP6 = new person("Сидоров", "Николай", 8435150);
//-----внесение данных в вектор
        vectPtrsPers.push_back(ptrP1);
        vectPtrsPers.push_back(ptrP2);
        vectPtrsPers.push_back(ptrP3);
        vectPtrsPers.push_back(ptrP4);
        vectPtrsPers.push_back(ptrP5);
        vectPtrsPers.push_back(ptrP6);
//-----вывод вектора
        cout << "\n\nИсходный вектор:";
        for_each(vectPtrsPers.begin(),
                vectPtrsPers.end(), displayPerson());
//-----сортировка указателей
        sort(vectPtrsPers.begin(), vectPtrsPers.end());
        cout << "\n\nУказатели отсортированы:";
//-----вывод вектора
        for_each(vectPtrsPers.begin(),
                vectPtrsPers.end(), displayPerson());
//-----сортировка персональных данных
        sort(vectPtrsPers.begin(),
                vectPtrsPers.end(), comparePersons());
        cout << "\n\nПерсональные данные отсортированы :";
//-----вывод вектора
        for_each(vectPtrsPers.begin(),
                vectPtrsPers.end(), displayPerson());
        while (!vectPtrsPers.empty())
        {
            delete vectPtrsPers.back(); //удаление персоны
            vectPtrsPers.pop_back();    //выталкивание указателя
        }
        cout << endl;
        system("pause");
        return 0;
}

```

Вот как выглядит на экране результат работы программы:

```

Исходный вектор:
    Иванов    Фёдор    т-ф : 8435150
    Петров    Борис    т-ф : 3327563
    Фёдоров    Пётр    т-ф : 6946473
    Мартынов    Валерий    т-ф : 4157300
    Кузин    Евгений    т-ф : 9207404
    Сидоров    Николай    т-ф : 8435150

Указатели отсортированы:
    Кузин    Евгений    т-ф : 9207404
    Сидоров    Николай    т-ф : 8435150
    Иванов    Фёдор    т-ф : 8435150
    Петров    Борис    т-ф : 3327563
    Фёдоров    Пётр    т-ф : 6946473
    Мартынов    Валерий    т-ф : 4157300

Персональные данные отсортированы :
    Иванов    Фёдор    т-ф : 8435150
    Кузин    Евгений    т-ф : 9207404
    Мартынов    Валерий    т-ф : 4157300
    Петров    Борис    т-ф : 3327563
    Сидоров    Николай    т-ф : 8435150
    Фёдоров    Пётр    т-ф : 6946473
Для продолжения нажмите любую клавишу

```

Вначале выводятся данные в исходном порядке, затем — некорректно отсортированные по указателям, наконец, правильно упорядоченные по именам людей.

Функциональный объект `comparePersons()`

Если мы возьмем версию алгоритма `sort()`, имеющую два аргумента

```
sort( vectPtrsPers.begin(), vectPtrsPers.end() );
```

то будут сортироваться только указатели по их адресам в памяти. Такая сортировка требуется крайне редко. Чтобы упорядочить объекты `person` по именам, мы воспользуемся алгоритмом `sort()` с тремя аргументами. В качестве третьего аргумента будет выступать функциональный объект `comparePersons()`.

```
sort( vectPtrsPers.begin(), vectPtrsPers.end(), comparePersons() );
```

Что касается нашего собственного функционального объекта `comparePersons()`, то его мы определяем в программе таким образом:

```
// функциональный объект для сравнения содержимого
//указателей на объекты person
class comparePersons
{
```

```
public:
    bool operator()(const person* ptrP1, const person* ptrP2) const { return *ptrP1 <
*ptrP2; }
};
```

При этом `operator()` имеет два аргумента, являющихся указателями на персональные данные, и сравнивает значения их содержимого, а не просто значения указателей.

Функциональный объект `displayPerson()`

Мы несколько необычным образом осуществляем вывод содержимого контейнера. Вместо простой итерации с поэлементным выводом мы используем функцию `for_each()` с собственным функциональным объектом в качестве третьего аргумента.

```
for_each( vectPtrsPers.begin(), vectPtrsPers.end(), displayPerson() );
```

Это выражение приводит к тому, что функциональный объект `displayPerson()`

выводится для каждого элемента данных в векторе. Вот как выглядит объект

```
displayPerson():
//функциональный объект для вывода персональных данных,
//храняемых в указателях class displayPerson
{
public:
    void operator() (const person* ptrP) const { ptrP->display(); }
};
```

Таким образом, с помощью одного-единственного вызова функции на экран выводится содержимое всего вектора!

Функциональные объекты и поведение контейнеров

Функциональные объекты реально могут изменять поведение контейнеров. В программе `SORTPTRS` мы показали это. Например, с их помощью можно осуществлять сортировку данных во множестве указателей не по адресам последних, а по их содержимому, то есть сортировать реальные объекты, а не ссылки на них. Соответственно, при определении контейнера необходимо определить подходящий функциональный объект. И тогда не понадобятся никакие алгоритмы `sort()`.

Резюме

Итак, мы кратко обсудили вопросы, связанные с основами STL. Тем не менее, мы затронули большую часть тем, связанных с этим мощным инструментом C++. После внимательного изучения предложенного материала можно смело приступать к практическому применению STL. Для более полного понимания возможностей Стандартной библиотеки шаблонов необходимо ознакомиться со специализированными изданиями, посвященными этой теме.

Мы выяснили, что STL состоит из трех основных компонентов: контейнеров, алгоритмов и итераторов. Контейнеры подразделяются на две группы: последовательные и ассоциативные. Последовательными являются векторы, списки и очереди с двусторонним доступом. Ассоциативные контейнеры — это, прежде всего, множества и отображения, а также тесно связанные с ними мультимножества и мультиотображения. Алгоритмы выполняют определенные операции над контейнерами, такие, как сортировка, копирование и поиск. Итераторы представляют собой разновидность указателей, применяющихся к элементам контейнеров, также они являются связующим звеном между алгоритмами и контейнерами.

Не все алгоритмы могут работать со всеми контейнерами. Итераторы используются для того, в частности, чтобы удостовериться, что данный алгоритм подходит данному контейнеру. Итераторы определяются только для некоторых видов контейнеров и могут являться аргументами алгоритмов. Если итератор контейнера не соответствует алгоритму, возникает ошибка компилятора.

Входные и выходные итераторы используются для подключения контейнеров непосредственно к потокам ввода/вывода, в результате этого возникает возможность работы контейнеров непосредственно с устройствами ввода/вывода! Специализированные итераторы дают возможность проходить контейнеры в обратном направлении, а также изменять поведение некоторых алгоритмов, например, заставляя их вставлять данные, а не перезаписывать их.

Алгоритмы являются шаблонами функций, которые могут работать со многими типами контейнеров. У контейнера есть свои собственные специфические методы. В некоторых случаях одна и та же функция может существовать одновременно и в виде алгоритма, и в виде метода.

Контейнеры и алгоритмы STL работают с данными как встроенных так и определяемых пользователем типов, при наличии соответствующих перегружаемых операций.

Поведение отдельных алгоритмов, таких, как `find_if()`, может быть

изменено с помощью функциональных объектов. Последние реализуются с помощью классов, содержащих только один оператор вызова функции ().