

# **Адаптеры. Работа со словарём**

# Адаптеры контейнеров

Рассмотрим специализированные последовательные контейнеры – *стек, очередь и очередь с приоритетами.*

Они не являются самостоятельными контейнерными классами, а реализованы на основе рассмотренных выше классов (вектора, двусторонней очереди и списка), поэтому они называются *адаптерами контейнеров.*

## Стек (stack) –

структура данных, которая допускает только две операции, изменяющие ее размер: **push** (добавление элемента в конце) и **pop** (удаление элемента в конце). Стек работает по принципу «последний пришел – первый ушел» (**LIFO** от английского Last In – First Out).

Кроме **push** и **pop** для стека определены также функции-члены **empty** и **size**, имеющие обычное значение, и **top** (вместо **back**) для доступа к последнему элементу.

Стек может быть реализован с помощью каждого из трех последовательных контейнеров STL: вектора, двусторонней очереди и списка. =>

Стек – это не новый тип контейнера, а особый вариант вектора, двусторонней очереди либо списка, отсюда и происхождение термина **адаптер контейнера**.

## Стек (stack)

В качестве примера используем стек для чтения последовательности целых чисел и отображения их в обратном порядке. Любой нецифровой символ будет признаком конца ввода. В следующей программе ***стек реализован вектором***, но программа также будет работать, если мы заменим всюду ***vector*** на ***deque*** или ***list***. Кроме того, программа показывает, как работают функции-члены ***empty***, ***top*** и ***size***.

```
#include <iostream>
```

```
#include <vector>
```

```
#include <stack>
```

```
.....
```

```

int main1()
{ stack <int, vector<int> > S; int x;
cout << "Enter some integers, followed by a letter:\n";
while (cin >> x) S.push(x);
while (!S.empty())
    { x = S.top();
    cout << "Size: " << S.size()
    << " Element at the top: " << x << endl; S.pop();
    }
return 0;
} // Шаблон stack имеет два параметра:
// stack <int, vector<int> > S;

```

Enter some integers, followed by a letter:

10 20 30 A

Size: 3 Element at the top: 30

Size: 2 Element at the top: 20

Size: 1 Element at the top: 10

# Стек (stack)

Для стеков мы **не** можем использовать итераторы, а также **не** можем получить доступ к произвольному элементу стека без изменения его размера. Стек определяет операторы присваивания (**=**) и сравнения (**==** и **<**). Отсюда следует, что для стеков можно использовать также остальные четыре оператора сравнения. Оператор **<** осуществляет лексикографическое сравнение, как показывает следующая программа:

```
// stackcmp.cpp: Сравнение и присваивание для стеков.
```

```
#include <iostream>
```

```
#include <vector>
```

```
#include <stack>
```

```
.....
```

```
int main2()
```

```
{ stack <int, vector<int> > S, T, U;  
S.push(10); S.push(20); S.push(30);  
cout << "Pushed onto S: 10 20 30\n";  
T = S;  
cout << "After T = S; we have ";  
cout << (S == T ? "S == T" : "S != T") << endl;  
U.push(10); U.push(21);  
cout << "Pushed onto U: 10 21\n";  
cout << "We now have ";  
cout << (S < U ? "S < U" : "S >= U") << endl;  
return 0;  
}
```

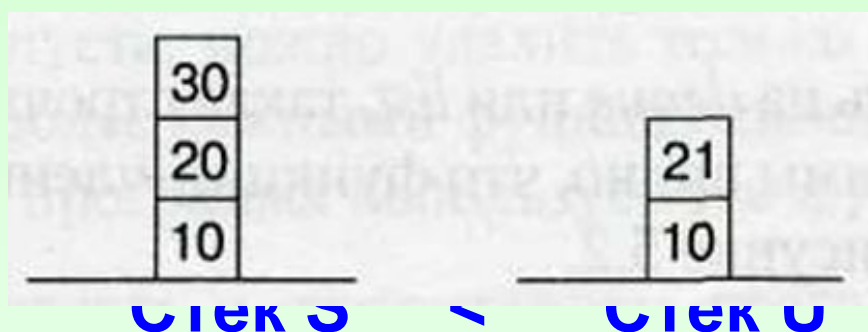
## Вывод программы:

Pushed onto S: 10 20 30

After  $T = S$ ; we have  $S == T$

Pushed onto U: 10 21

We now have  $S < U$



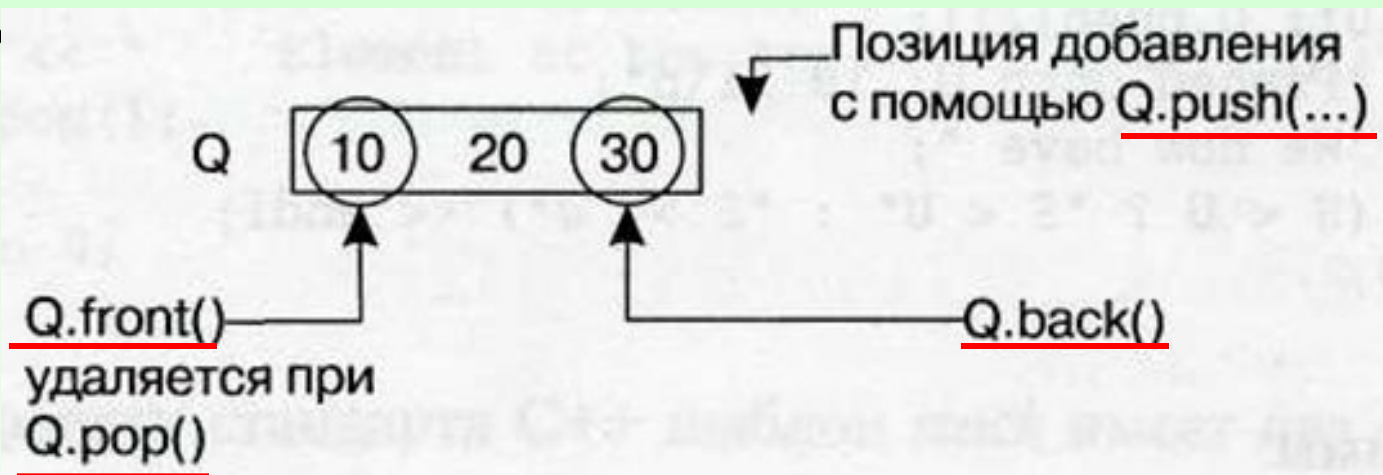
Лексикографическое  
сравнение стеков  
(последнее сравнение)

Первыми сравниваются элементы **внизу стека**. Поскольку оба они равны 10, происходит сравнение следующих за ними элементов 20 и 21. В нашем примере  $S < U$ , потому что  $20 < 21$ . Это сравнение выполняется таким же образом, как и сравнение строк, только для строк мы начинаем сравнение с первых символов, а для стека – с нижних элементов.



## Очередь (queue) –

структура данных, в которую можно добавлять элементы с одного конца, – сзади, и удалять с другого конца, – спереди. Мы можем узнать и изменить значения элементов в начале и в конце очереди



В отличие от стека **очередь** нельзя представить с помощью **вектора**, поскольку у вектора отсутствует операция **pop\_front**. Например, нельзя написать

```
queue <int, vector<int> > Q; // Ошибка!!!
```

Но если **vector** заменить на **deque** или **list**, такая строчка станет допустимой. Функции-члены **push** и **pop** работают так, как показано на рисунке.

# Использование очереди.

## Функции *push*, *pop*, *back* и *front*

```
#include <iostream>
#include <list>
#include <queue>
    int main3()
{ queue <int, list<int> > Q;
Q.push(10); Q.push(20); Q.push(30);
cout << "After pushing 10, 20 and 30:\n";
cout << "Q.front() = " << Q.front() << endl;
cout << "Q.back() = " << Q.back() << endl;
Q.pop() ;
cout << "After Q.pop():\n";
cout << "Q.front() = " << Q.front() << endl;
return 0;
}
```

## Очередь – продолжение программы

### *Вывод программы:*

After pushing 10, 20 and 30:

`Q.front() = 10`

`Q.back() = 30`

After `Q.pop()`:

`Q.front() = 20`

Функции-члены *empty* и *size* класса *queue* аналогичны этим функциям для класса *stack*, так же как и операторы присваивания и сравнения. Сравнение начинается с передних элементов; если они равны, происходит сравнение следующих, и так далее.

## Очередь с приоритетами (priority queue) -

структура данных, из которой, если она не пуста, можно удалить только наибольший элемент. Как и для стеков, наиболее важными функциями-членами являются *push*, *pop* и *top*.

// Очередь с приоритетами: *push*, *pop*, *empty* и *top*.

```
#include <iostream>          #include <vector>
#include <functional>        #include <queue>
int main4()
{ priority_queue <int, vector<int>, less<int> > P;
  int x;
  P.push(123); P.push(51); P.push(1000); P.push(17);
  while (!P.empty())
    { x = P.top();
      cout << "Retrieved element: " << x << endl;
      P.pop(); }
  return 0;
}
```

В этой программе числа следуют в нисходящем порядке:

Retrieved element: 1000

Retrieved element: 123

Retrieved element: 51

Retrieved element: 17

Поскольку требуется проводить сравнение элементов, шаблон *priority\_queue* имеет третий параметр, как видно из определения очереди с приоритетами *P*:

```
priority_queue <int, vector<int>, less<int> > P;
```

Если требуется извлекать элементы в порядке возрастания, мы можем просто заменить *less<int>* на *greater<int>*.

Существует возможность задания любого правила упорядочения элементов.

Рассмотрим пример, в котором элементы будут извлекаться **по порядку возрастания последних цифр в десятичном представлении целых чисел**, хранящихся в очереди с приоритетами:

```
#include <iostream>
#include <vector>
#include <functional>
#include <queue> .....
class CompareLastDigits {
public:
bool operator()(int x, int y)
{ return x % 10 > y % 10; } };
```

Необходимо использовать **функциональный объект**, поскольку шаблон **priority\_queue** требует в качестве третьего параметра тип. Этот тип определяет идентификатор **CompareLastDigits**. Сравнение  $x \% 10 > y \% 10$  содержит оператор **>**, в результате чего элемент с наименьшей последней цифрой предшествует другим элементам.

```

int main5()
{ priority_queue <int, vector<int>, CompareLastDigits> P;
  int x;
  P.push(123); P.push(51); P.push(1000); P.push(17);
  while (!P.empty())
    { x = P.top();
      cout << "Retrieved element: " << x << endl;
      P.pop(); }
  return 0; }

```

**P.top()** указывает на элемент, последняя цифра которого не больше последних цифр других элементов.

Вывод этой программы содержит добавленные целые числа 123, 51, 1000, 17 в порядке возрастания их последних цифр (**0 < 1 < 3 < 7**):

```

Retrieved element: 1000
Retrieved element: 51
Retrieved element: 123
Retrieved element: 17

```

## Пары и сравнения

Чтобы использовать словари и словари с дубликатами более интересным способом, воспользуемся шаблонным классом *pair* (пара).

```
#include <utility> // В заголовочном файле <utility> //  
описывается шаблон pair для хранения пары  
// «ключ-элемент»
```

```
int pairs()
```

```
{ pair<int, double> P(123, 4.5), Q = P;
```

```
Q = make_pair (122,4.5);
```

```
cout << "P: " << P.first << " " << P.second << endl;
```

```
cout << "Q: " << Q.first << " " << Q.second << endl;
```

```
if (P > Q) cout << "P > Q\n";
```

```
++Q.first;
```

```
cout << "After ++Q.first: ";
```

```
if (P == Q) cout << "P == Q\n";
```

```
return 0; }
```



## Пары и сравнения

Шаблон *pair* имеет два параметра, представляющих собой типы членов структуры *pair: first u second*. Для пары определено два конструктора:

- Один должен получать два значения для инициализации элементов,
- Второй (конструктор копирования) – ссылку на другую пару.

Конструктора по умолчанию у пары нет, => при создании объекта ему требуется присвоить значение явным образом.

Для присваивания значения паре можно использовать функцию *make\_pair*:

```
Q = make_pair(122, 4.5);
```

Но можно присвоить значение *Q*, написав:

```
Q = pair<int, double>(122; 4.5);
```

## Пары и сравнения

Для пары определены проверка на равенство (**==**) и операция сравнения (**<**), все остальные операции генерируются на основе этих двух автоматически. Пара **P** меньше пары **Q**, если **P.first < Q.first** или **P.first == Q.first && P.second < Q.second**.

Например, для любых двух пар **P** и **Q**:

**(122, 5.5) < (123, 4.5)**

**(123, 4.5) < (123, 5.5)**

**(123, 4.5) == (123, 4.5)**

В программе **pairs.cpp** сначала мы имеем **P > Q**, но после увеличения **Q.first** на единицу **P == Q**.

**P: 123 4.5**

**Q: 122 4.5**

**P > Q**

**After ++Q.first: P == Q**

# Сравнения

Когда мы пишем операторы сравнения для наших собственных типов, нам необходимо определить только **==** и **<**. Четыре остальных оператора **!=**, **>**, **<=** и **>=** автоматически определяются в STL с помощью следующих четырех шаблонов:

```
template <class T1, class T2>
```

```
inline bool operator!=(const T1 &x, const T2 &y)
```

```
{ return !(x == y) ; }
```

```
template <class T1, class T2>
```

```
inline bool operator>(const T1 &x, const T2 &y)
```

```
{ return y < x; }
```

```
template <class T1, class T2>
```

```
inline bool operator<=(const T1 &x, const T2 &y)
```

```
{ return !(y < x); }
```

## Сравнения

```
template <class T1, class T2>  
inline bool operator>=(const T1 &x, const T2 &y)  
{ return !(x < y) ; }
```

Как видно из примера, эти четыре достаточно общих шаблона определяют *!=*, *>*, *<=* и *>=* через *==* и *<*. Нам не нужно писать эти шаблоны самостоятельно, поскольку они находятся в заголовке *functional*, который включается по умолчанию всякий раз, когда мы используем STL.

-----

Заголовочный файл *<utility>* при использовании *<map>* или *<set>* подключается автоматически.

## Пример со словарём

Словарь содержит пары  $(k, d)$ , где  $k$  – ключ, а  $d$  – сопутствующие данные.

Как и для последовательного контейнера, для ассоциативного контейнера будем использовать итератор  $i$ ; в этом случае выражение  $*i$  будет обозначать пару, в которой  $(*i).first$  является ключом, а  $(*i).second$  – сопутствующими данными. Например, с помощью итератора  $i$  напечатаем все содержимое словаря (ключи в восходящем порядке), применив следующий цикл **for**.

```
for (i = D.begin(); i != D.end(); i++)  
    cout << setw(9)  
    << (*i).second <<" "  
    << (*i).first << endl;
```

## Пример со словарём

Заметим, что здесь выводим *(\*i).second* перед *(\*i)first*, так что не нужно планировать, сколько позиций зарезервировать для имен в выводе:

54321 Papadimitrou, C.

12345 Smith, J.

С таким форматом также удобнее работать при вводе, поскольку в этом случае мы можем прочесть число, пробел и текст до конца строки. Но следует помнить, что этот текст является ключом, хотя и расположен в конце строки.

Рассмотрим программу *«Телефонный справочник»*, она будет иметь несложный интерфейс.

Данные будут считываться из файла *phone.txt*.  
Строчки текста в файле включают номер телефона, один пробел и имя, в перечисленном порядке.

# Пример интерфейса

## Пример команды

## Значение

- ?Johnson, J.      Показать телефонный номер абонента *Johnson, J.*
- /Johnson, J.      Удалить запись об абоненте *Johnson, J.* из книги
- !66331 Peterson, K.      Добавить абонента *Peterson, K.* с номером 66331
- \*      Показать всю телефонную книгу
- =      Записать телефонную книгу в файл *phone.txt*
- #      Выход

Имена являются ключами, хотя и следуют после номеров телефонов.

# Приложение, использующее класс *map* (словарь): Телефонный справочник (для VS 2013, UNICODE)

```
#include <iostream>
```

```
#include <fstream>
```

```
#include <iomanip>
```

```
#include <stdlib.h>
```

```
#include <map>
```

```
#include <locale>
```

```
#include <codecvt>
```

```
#include <string>
```

```
#include <cstdlib>
```



# Приложение, использующее класс map (словарь):

## Телефонный справочник

Определение словаря задаётся таким образом:

```
typedef map<CString, long, compare_m>  
directype;
```

```
class compare_m // функциональный объект
```

```
{
```

```
public:
```

```
bool operator()(const CString s, const CString t)const
```

```
{
```

```
    return (s < t);
```

```
}
```

```
};
```

## Основные функции, используемые в приложении:

- **void ReadInput (directype &D)** – чтение данных из файла;
  - **void ShowCommands ()** – создание меню;
  - **void ProcessCommands(directype &D)** – реализация основных команд приложения.
- 

### ***Commands (Меню):***

? name :find phone number

/name :delete

!number name :insert (or update)

\* :list whole phonebook

= :save in file

# :exit

## Entries read from file phone.txt:

54321 Smith, P.

12345 Johnson, J.

!19723 Shaw, A.

\*

12345 Johnson, J

19723 Shaw, A.

54321 Smith, P.

/Johnson, J.

\*

19723 Shaw, A.

54321 Smith, P.

?Shaw, A.

Number: 19723

=

#

Из файла прочитали

две записи.

– ввод новой записи

– вывод справочника

В справочнике стало

три записи.

= удаление записи

– вывод справочника

После удаления осталось

две записи.

– поиск по ключу (по фамилии)

Результат поиска.

– сохранение в файле

– выход из приложения

## **void ReadInput (directype &D) - 1**

```
{  
int i,k;  
long nr;  
CStdioFile f;  
if (!f.Open(_T("phone.txt"), CFile::modeRead))  
{  
    wcout << _T("File phone.txt is not opened!\n");  
    return;  
}  
CString s,buf;  
    wcout << _T("Entries read from file phone.txt:\n");
```

## void ReadInput (directype &D) - 2

```
while (f.ReadString(s))
```

```
{
```

```
    nr = _wtoi(s);
```

```
    k = wcslen(s);
```

```
    if (k < 2) break;
```

```
    for (i = 0; i < k; i++) // пропустить пробел
```

```
    { if (!iswalph(s[i])) continue;
```

```
        buf = s.Right(k - i); break;
```

```
    }
```

```
wcout << setw(9) << nr << " " << (const wchar_t*)buf <<
```

```
endl;    D[buf] = nr;
```

```
}
```

```
f.Close();
```

```
}
```

## void ProcessCommands(directype &D) - 1

{

wofstream ofstr;

CStdioFile f;

long nr;

TCHAR ch;

wstring buf;

CString s;

directype::iterator i;

for (;;)

{

wcin >> ch; // Пропустить любой

// непечатаемый символ и прочесть ch.

## void ProcessCommands(directype &D) - 2

```
switch (ch)
{
    case '?': case '/':    // найти или удалить:
        getline(wcin, buf);
        s = buf.c_str();
        i = D.find(s);
        if (i == D.end()) wcout << L"Not found. \n";
        else                // Ключ найден.
            if (ch == '?') // Команда 'Найти'
                wcout << L"Number: " << (*i).second << endl;
            else            // Команда 'Удалить'
                { D.erase(i); }
        break;
}
```

## void ProcessCommands(directype &D) - 3

```
case '!':      // добавить (или обновить)
    wcin >> nr;
    if (wcin.fail())
    {
        wcout << L"Usage: !number name\n";
        wcin.clear();
        getline(wcin, buf);    break;
    }
    wcin.get();    // пропустить пробел
    getline(wcin, buf);
    s = buf.c_str();
    i = D.find(s);
```



## **void ProcessCommands(directype &D) - 4**

```
if (i == D.end())  
{ D[s] = nr; }  
else (*i).second = nr; break;  
case '*':  
    for (i = D.begin(); i != D.end(); i++)  
        wcout << setw(9) << (*i).second << L" "  
            << (const wchar_t*)((*i).first) << endl;  
    break;  
case '=':  
    if (f.Open(_T("phone.txt"), CFile::modeWrite))  
    {  
        CString s, buf;
```

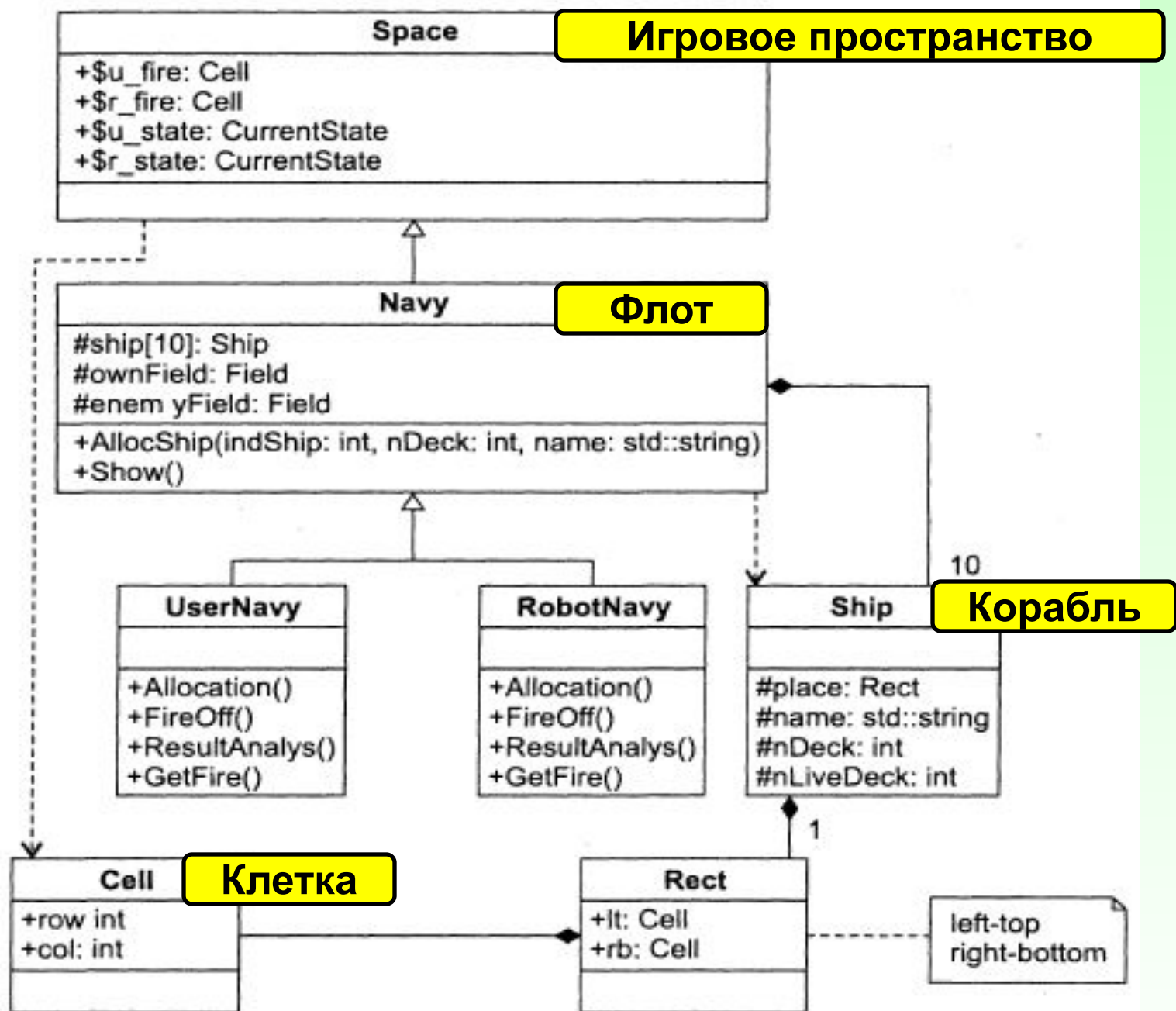
## **void ProcessCommands(directype &D) - 5**

```
for (i = D.begin(); i != D.end(); i++)  
{ s.Format(_T(" %d %s\n"), (*i).second, (*i).first);  
  f.WriteString(s); }  
f.Close();  
} break;  
case '#':  
  break;  
default:  
wcout << L"Use: * (list), ? (find), = (save), "  
  L"/ (delete), ! (insert), or # (exit).\n";  
getline(wcin, buf);  
  break;  
} if (ch == '#') break; } }
```

**int map2()**

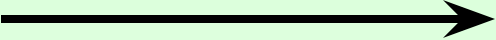
```
{  
    directype D;  
    ReadInput(D);  
    ShowCommands();  
    ProcessCommands(D);  
    return 0;  
}
```


# Морской бой (диаграмма классов)



## Отношения на диаграмме:

- зависимости,
- обобщения,
- ассоциации.

**Зависимость** – хотим показать, что один класс использует другой. 

**Обобщение** – отношение типа «является», наследование, объекты класса-потомка могут использоваться всюду, где встречаются объекты класса-родителя, но не наоборот. 

**Ассоциация** – описывает совокупность связей между объектами \_\_\_\_\_ . Частный случай -

**Композиция** – четко выражены отношения владения, причем время жизни частей и целого совпадают. 