Лекция 6

Динамические структуры данных

Динамические структуры данных – память выделяется и освобождается по мере необходимости.

Динамическое распределение памяти - память под отдельные элементы выделяется в момент, когда они "начинают существовать" в процессе выполнения программы, а не во время компиляции. Компилятор выделяет фиксированный объем памяти для хранения адреса динамически размещаемого элемента, а не самого элемента.

Динамическая структура данных:

- не имеет имени;
- ей выделяется память в процессе выполнения программы;
- количество элементов структуры может не фиксироваться;
- размерность структуры может меняться в процессе выполнения программы;
- в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

Каждой динамической структуре данных сопоставляется статическая переменная типа указатель (ее значение – адрес этого объекта), посредством неё осуществляется доступ к динамической структуре.

Динамические величины не требуют описания в программе (при компиляции память под них не выделяется).

Во время компиляции память выделяется только под статические величины. Указатели – это статические величины, поэтому они требуют описания.

Необходимость в динамических структурах данных возникает в случаях:

- Используются переменные большого размера, необходимые в одних частях программы и не нужные в других.
- В процессе работы нужен массив, список или иная структура, размер которой изменяется в широких пределах и трудно предсказуем.
- Размер обрабатываемых данных превышает объем сегмента данных.

Динамические структуры характеризуются отсутствием физической смежности элементов структуры в памяти, непостоянством и непредсказуемостью размера (числа элементов) структуры в процессе ее обработки.

Элементы располагаются по непредсказуемым адресам памяти, поэтому адрес элемента не может быть вычислен из адреса начального или предыдущего элемента.

Для связи между элементами используются указатели - связное представление.

Достоинства – возможность обеспечения изменчивости структур:

- размер структуры ограничивается только доступным объемом машинной памяти;
- при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей;
- большая гибкость структуры.

Основной недостаток:

- на поля, содержащие указатели, расходуется дополнительная память;
- доступ к элементам может быть менее эффективным по времени.

Поэтому связное представление практически не применяется в задачах с векторами или массивами (с доступом по номеру элемента), но применяется в задачах с другой исходной информации доступа (таблицы, списки, деревья и т.д.).

Порядок работы с динамическими структурами данных:

- создание (отвести место в динамической памяти);
- работа при помощи указателя;
- удаление (освободить занятое структурой место).

Классификация динамических структур данных:

- Списки: однонаправленные (односвязные), двунаправленные, циклические;
- Стек;
- Дек;
- Очередь;
- Бинарные деревья.

Отличаются способом связи элементов и/или допустимыми операциями.

Динамическая структура может занимать несмежные участки оперативной памяти.

Элемент динамической структуры состоит (по крайней мере) из двух полей (рис. 5.2):

- информационное (поле данных) – для размещения данных;
- адресное поле – поле типа указатель для связывания элементов.

Информационных и адресных полей может быть одно и несколько.

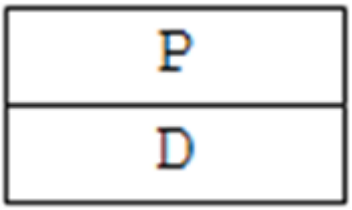


Рис. 5.2. Поля элемента динамической структуры: поле P – указатель, поле D - данные

Объявление элемента динамической структуры :

```
struct имя_типа {
   информационное поле;
   адресное поле;
   }
```

Например:

```
struct TNode {
    int Data;           //информационное поле
    TNode *Next;  //адресное поле
}
```

Пример – динамическая структура из3 4 элементов (рис. 5.3).

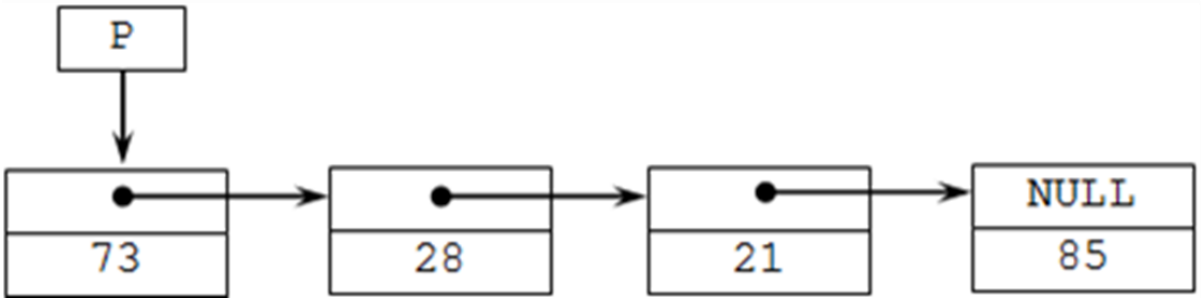


Рис. 5.3. Динамическая структура из 4 элементов

Элемент динамической структуры в каждый момент может либо существовать, либо отсутствовать в памяти, поэтому его называют динамическим.

Элементами динамической структуры являются динамические переменные.

Средство доступа к элементам динамических структур - указатель (адрес) на место их текущего расположения в памяти.

Указатель содержит адрес объекта в динамической памяти. Адрес формируется из двух слов: адрес сегмента и смещение. Сам указатель является статическим объектом и расположен в сегменте данных (рис. 5.4).

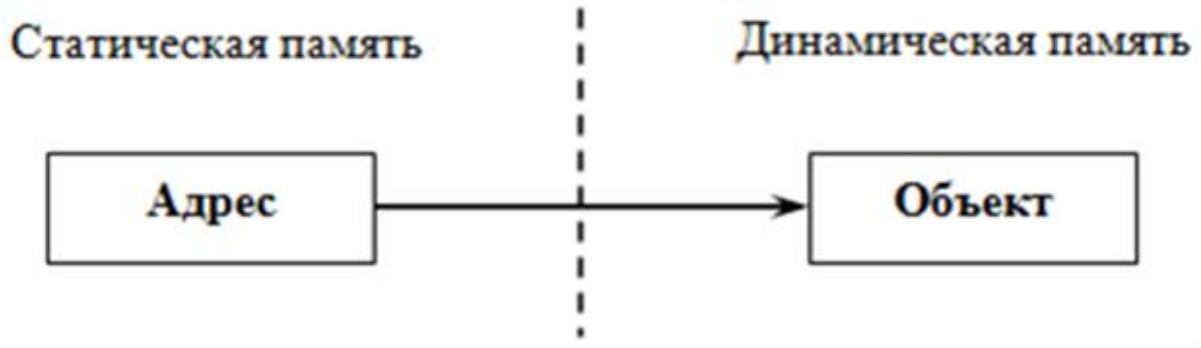


Рис. 5.4. Доступ к динамическим структурам

Для обращения к динамической структуре достаточно хранить в памяти адрес 1-ого элемента. Каждый элемент хранит адрес следующего за ним элемента, можно, двигаясь от начального элемента по адресам, получить доступ к любому элементу данной структуры.

Доступ к данным - с помощью операции "стрелка" (->) - косвенный выбор элемента объекта, адресуемого указателем.

Операция "стрелка" (->) двуместная - для доступа к элементу, задаваемому правым операндом, той структуры, которую адресует левый операнд.

Левый операнд - указатель на структуру, правый – имя элемента этой структуры.

```
p->Data;

p->Next;
```

```
struct Node {char *Name;
             int Value;
             Node *Next
            };
Node *PNode; //объявляется указатель

PNode = new Node; //выделяется память

PNode->Name = "STO"; //присваиваются значения
PNode->Value = 28;
PNode->Next = NULL;

delete PNode; // освобождение памяти
```

Список - упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения, исключения.

Список, отражающий отношения соседства между элементами – линейный список. Длина списка равна числу элементов в списке, список нулевой длины - пустой список. В списке элементы образуют цепочку. Для связывания элементов используют указатели.

В минимальном случае, любой элемент линейного списка имеет один указатель, который указывает на следующий элемент в списке или является пустым указателем (конец списка). Структура, элементами которой служат записи, связанные друг с другом с помощью указателей, хранящихся в самих элементах - связанный список.

В связанном списке элементы линейно упорядочены, порядок определяется не номерами, как в массиве, а указателями, входящими в состав элементов списка. Каждый список имеет особый элемент - указатель начала списка (голова списка), который обычно по содержанию отличен от остальных элементов. В поле указателя последнего элемента списка находится признак NULL - конец списка.

Основные списки:

- однонаправленные (односвязные);

- двунаправленные (двусвязные);

- циклические (кольцевые).

В основном они отличаются видом взаимосвязи элементов и/или допустимыми операциями.

Однонаправленный список - последовательность элементов, в каждом из которых хранится значение и указатель на следующий элемент списка. В последнем элементе указатель на следующий элемент равен NULL (рис. 5.5).

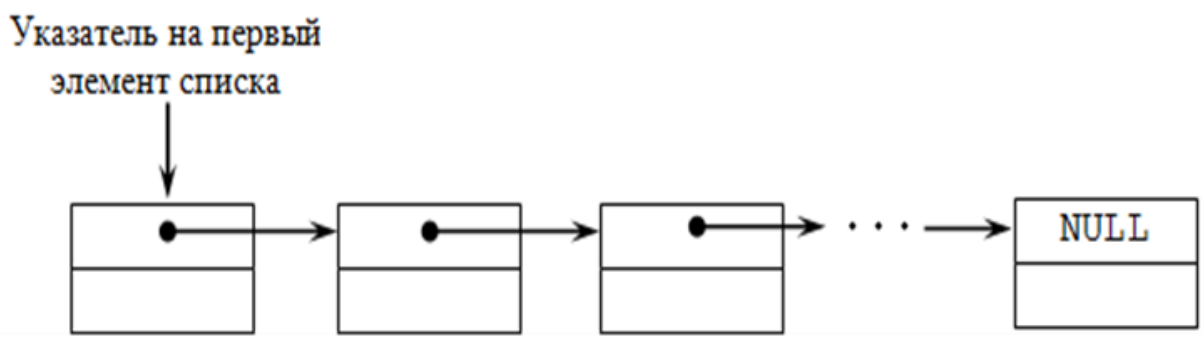


Рис. 5.5. Однонаправленный список

Описание элемента списка.

Инф. полей может быть несколько:

```
struct Node {
        int key;//информационное поле
        Node*next;//адресное поле
       };
```

Основные операции:

- создание
- печать (просмотр)
- вставка элемента в список (рис. 5.6);
- удаление элемента из списка (рис. 5.7);
- поиск элемента в списке
- проверка пустоты списка;
- удаление списка.

```
//создание однонаправленного списка (добавления в конец)
void Make_Single_List(int n,Single_List** Head){
 if (n > 0) {
  (*Head) = new Single_List();
  //выделяем память под новый элемент
  cout << "Введите значение ";
  cin >> (*Head)->Data;
  //вводим значение информационного поля
  (*Head)->Next=NULL;//обнуление адресного поля
   Make_Single_List(n-1,&((*Head)->Next));
 }
}
```

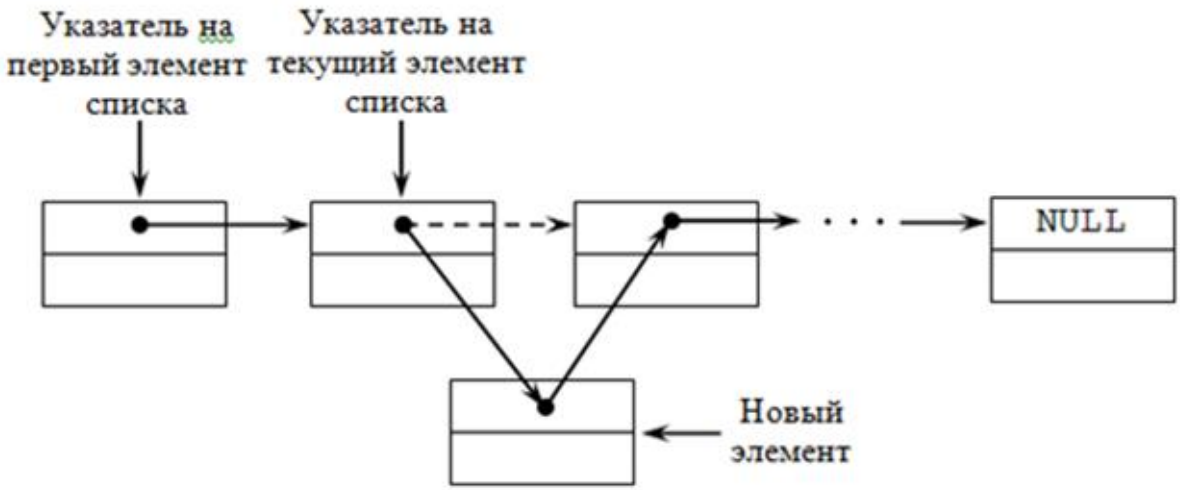


Рис. 5.6. Вставка элемента в однонаправленный список

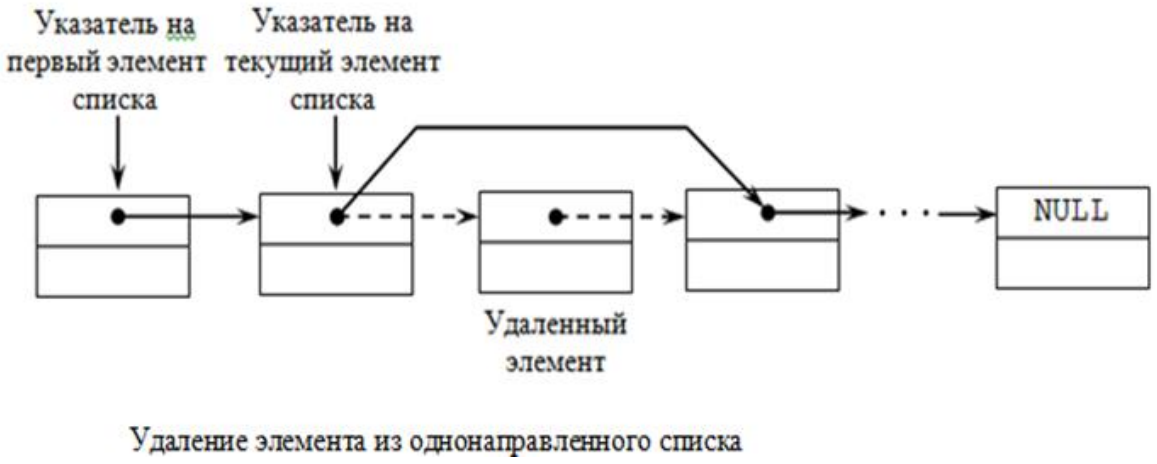


Рис. 5.7. Удаление элемента из однонаправленного списка

Двунаправленный (двусвязный) список – это структура данных, состоящая из последовательности элементов, каждый из которых содержит информационную часть и два указателя на соседние элементы. Два соседних элемента должны содержать взаимные ссылки друг на друга (рис. 5.8).

```
struct имя_типа {
        информационное поле;
        адресное поле 1;
        адресное поле 2;
        };
```

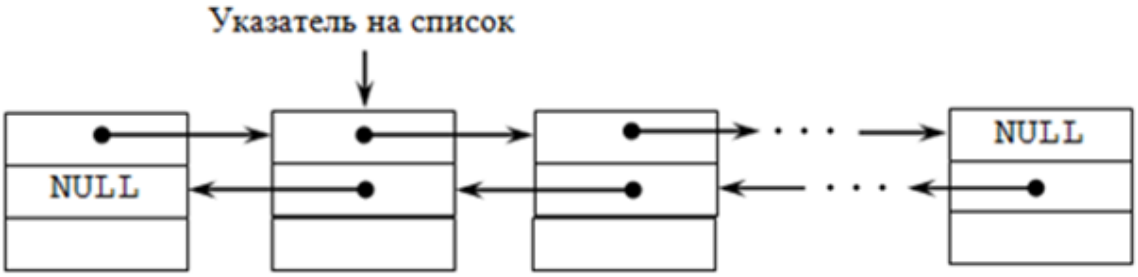
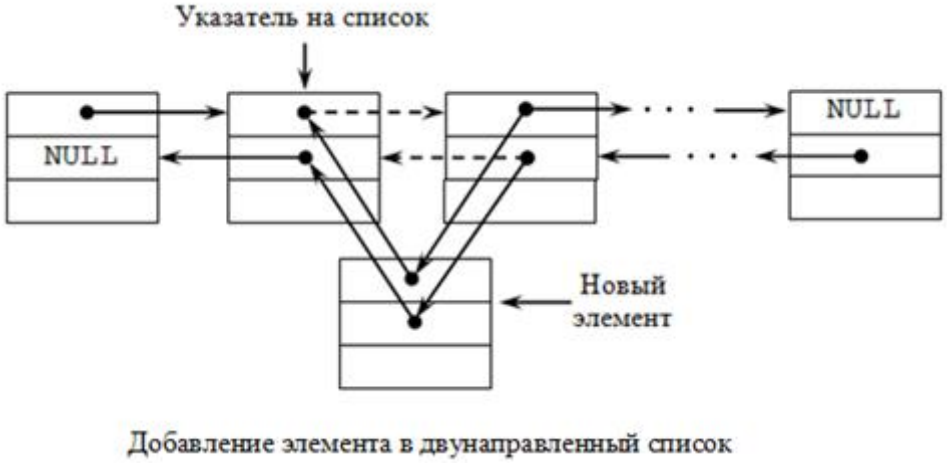


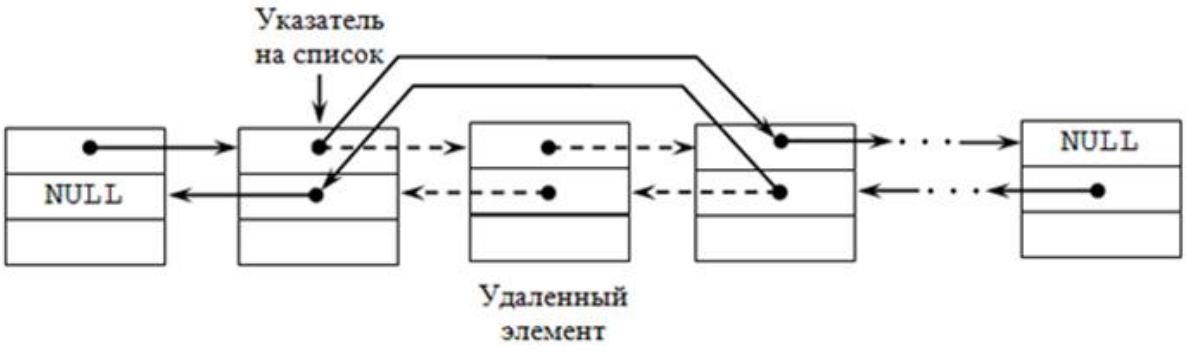
Рис. 5.8. Двунаправленный список

Иллюстрации добавления и удаления элемента в двунаправленном списке представлены на рис. 5.9. и 5.10 соответственно.



Добавление элемента в двунаправленный список

Рис. 5.9. Добавление элемента в двунаправленный список



Удаление элемента из двунаправленного списка

Рис. 5.10.Удаление элемента из двунаправленного списка

Стек и очередь – это частные случаи линейного списка.

Стек (англ. stack – стопка) – это структура данных, в которой новый элемент всегда записывается в ее начало (вершину) и очередной элемент также всегда выбирается из ее начала. Принцип доступа к элементам LIFO (Last Input – First Output) (рис. 5.11 и 5.12).

```
struct list {
        type pole1;
        list *pole2;
       } stack;
```

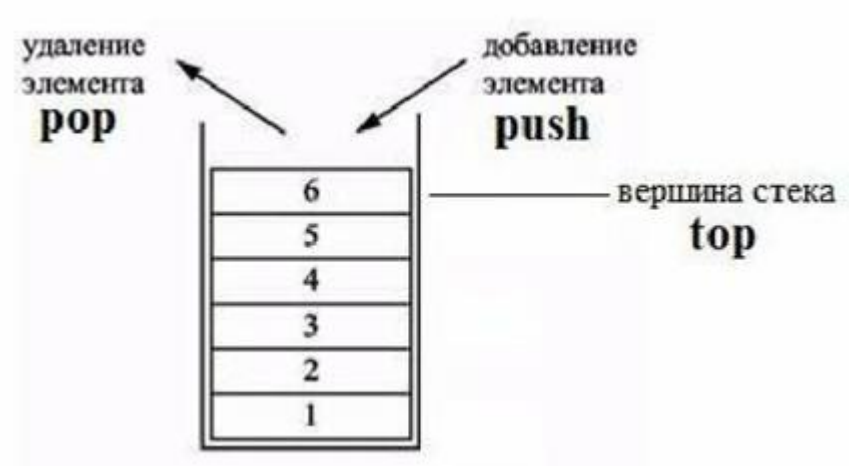


Рис. 5.11. Стек

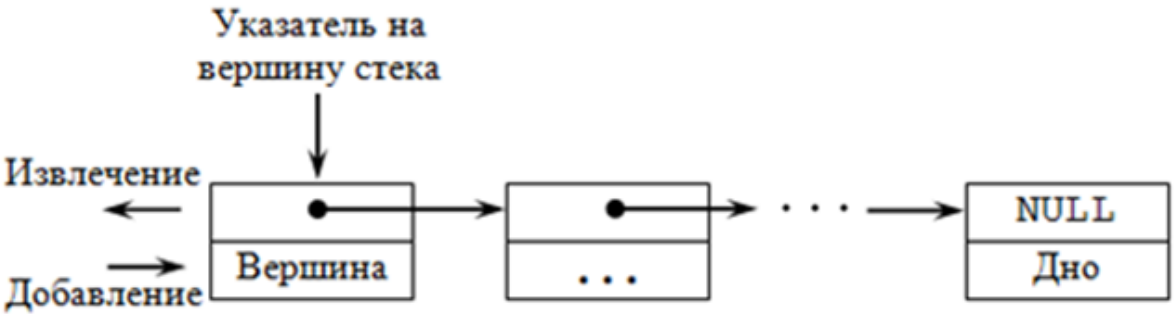


Рис. 5.12. Стек. Добавление и удаление элементов

Основные операции:

- создание стека;
- печать (просмотр) стека;

- добавление элемента в вершину стека;
- извлечение элемента из вершины стека;
- проверка пустоты стека;
- очистка стека.

Очередь – это структура данных, представляющая собой последовательность элементов, образованная в порядке их поступления. Принцип доступа к элементам FIFO (First Input – First Output). В очереди доступны два элемента - начало и конец очереди (рис. 5.13 и 5.14).

Поместить элемент можно только в конец очереди, а взять элемент только из ее начала.

```
struct list2 {
        type pole1;
        list2 *pole1, *pole2;
        }
```

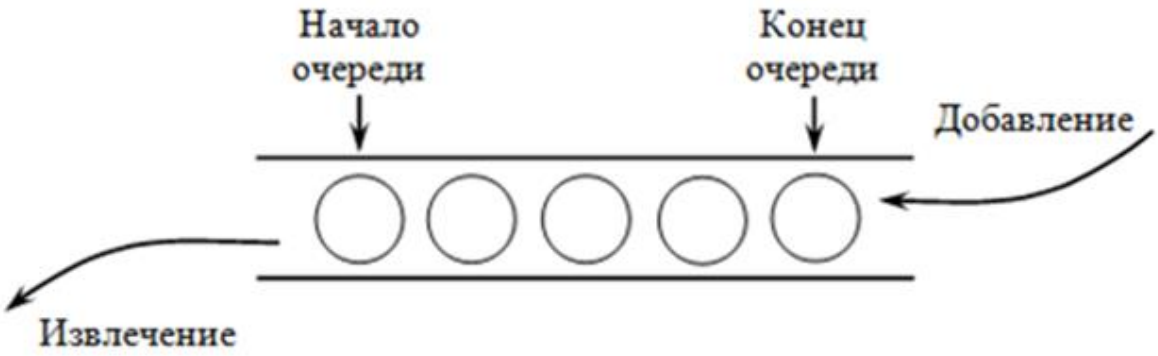
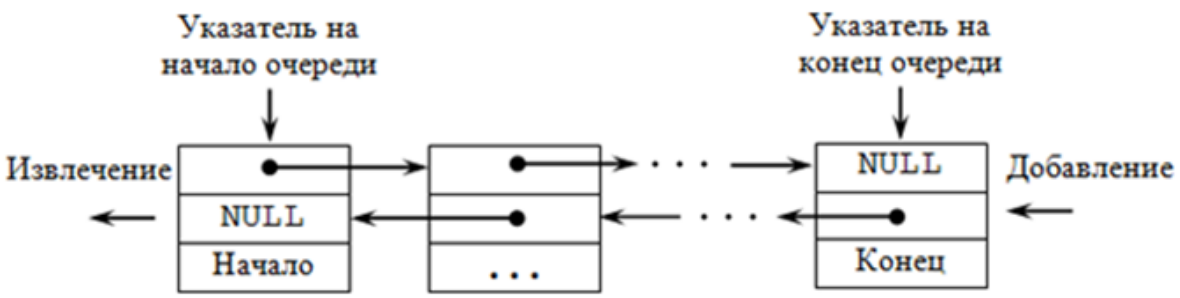


Рис. 5.13. Очередь



Очередь и ее организация

Рис. 5.14. Очередь и ее организация

Основные операции:

- создание очереди
- печать (просмотр) очереди
- добавление элемента в конец очереди
- извлечение элемента из начала очереди
- проверка пустоты очереди
- очистка очереди

Строки

Строка (в смысле языка C++) - набор символов.

Для чего мы можем его использовать, если есть тип данных char.

Переменные char полезны, когда хотим обрабатывать отдельные символы, но они чрезвычайно сложны, когда нам приходится иметь дело с данными, содержащими имена (ФИО, названия городов и т. д.) или просто текст.

Обработка таких данных в виде набора символов – это утомительно и медленно.

Проще обрабатывать все символы как единое целое, хранить, назначать и обрабатывать их одновременно.

Строка string несколько отличается от обычного типа данных (int, float).

(некоторые особенности объектно-ориентированного программирования).

```
#include <string>

string pet_name;
```

Объявили переменную с именем pet_name для хранения названий животных.

Важный факт: тип string не является встроенным типом (как int и др.)

Нужно поместить директиву #include в начало программы и запросить файл заголовка с именем string для включения во время компиляции.

Отсутствие директивы приведет к ошибке компиляции.

Слово “строка” не является ключевым словом в отличие от других имен типов (int, float)

Символы в строке пронумерованы с нуля 0.

1 способ – как и для других обычных типов, с помощью оператора присваивания, за которым следует строковый литерал (набор символов, заключенных в двойные кавычки).

Строки всегда используют кавычки, апострофы предназначены только для символьных литералов.

```
string pet_name = "Lassie";
```

Слово «строка» используют в двух разных значениях:

как имя типа (как в переменной типа string) и как сущность, состоящую из конечного числа символов

2 способ –Используем синтаксис, который напоминает вызов функции:

```
string pet_name("Lassie");
```

Существует ли реальная вызываемая функция с именем pet_name?

Ответ - “и да, и нет”. Да, потому что существует специализированная функция, отвечающая за создание строк, и эта функция вызывается каждый раз, когда вы хотите создать новую строку. Нет, потому что вам не разрешено напрямую вызывать функцию, как любую другую обычную функцию.

Обе формы инициализации одинаковы, и их результаты одинаковы. Можно использовать значение, хранящееся в любой строковой переменной, для инициализации вновь объявленной переменной:

```
string is_home = pet_name;

string has_returned(pet_name);
```

Допустимы обе формы (присваивающая и функциональная).

Оператор + (конкатенация) объединяет строки.

Объединение + не является коммутативным (a + b не всегда равно b + a) в отличие от сложения +.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string good = "Jekyll", bad = "Hyde";
  cout << good + " & " + bad << endl;
  cout << bad + " & " + good << endl;
}
Jekyll & Hyde
Hyde & Jekyll
```

Ограничение: оператор + не может объединять литералы. Он может объединить любую переменную с литералом, литерал с переменной, переменную с другой переменной. Программа содержит ошибку.

```
#include  <iostream>
#include  <string>
using namespace std;
int main()
{
  string s;
  s = "A" + "B";
  s = s + "C";
  s = "B" + s;
  cout << s << endl;
}
```

```
main.cpp: In function 'int main()':
main.cpp:10:13: error: invalid operands of types 'const char [2]' and 'const
char [2]' to binary 'operator+'
   s = "A" + "B";
```

Оператор + может использоваться в качестве оператора сокращения: +=

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string the question = "To be ";

  the question += "or not to be";
  cout << the question << endl;
}
To be or not to be
```

Вывод строк прост и не вызывает проблем.

С помощью потока cout содержимое строки будет передаваться посимвольно.

Ввод строк немного сложнее, так как поток cin обрабатывает пробелы как разделители, разграничивающие границы между данными. Могут возникнуть проблемы при вводе и сохранении строки, содержащую пробелы (и все белые символы).

Если ввести несколько слов с пробелами с клавиатуры, например,

To be or not to be

и нажать Enter, то на экране будет напечатано только первое слово (To).

Поток cin убежден, что вы ввели пробел, чтобы отметить конец строки.

Это не ошибка. Это сделано специально.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string line of types;

  cin >> line of types;
  cout << line of types << endl;
}
hello
hello

#include <iostream>
#include <string>
using namespace std;
int main()
{
  string line of types;
  getline(cin, line of types);
  cout << line of types << endl;
}
what is your name?
what is your name?
```

Если хотите ввести строку текста и обработать пробелы так же, как и любой другой символ - необходимо использовать функцию getline().

Эта функция получает/считывает все символы, введенные как есть, и не отдает предпочтения ни одному символу (кроме символа конца строки). В результате все символы, введенные до нажатия клавиши, будут введены в виде одной строки.

Строки можно сравнивать. Самый простой случай - когда надо проверить, содержат ли две переменные типа string одинаковые строки (==).

Операторы для сравнения строк:

== > < >= <= !=

Можно проверить, является ли одна из строк больше/меньше другой. Эти сравнения выполняются в алфавитном порядке, очевидно: "a" > "A", "z“>"a", но менее очевидно, что "a“>"1“.

```
#include <iostream>

#include <string>

using namespace std;

int main() {

    string secret = "abracadabra";

    string password;

    cout << "Enter password:" << endl;

    getline(cin, password);

    if (secret == password)

        cout << "Access granted" << endl;

    else

        cout << "Sorry";

}
```

Вывод:

Enter password:

mama

Sorry

Строки предлагают другой, более сложный, но и более мощный метод сравнения.

При классическом подходе у нас есть некоторые данные и набор функций, работающих с этими данными. Способ инициирования обработки данных: function(data)

В ОО подходе - иная терминология: данные и функции встроены в объект. Данные - это свойство объекта или переменная-член, функции - это методы объекта или функции-члены.

Если хотим, чтобы определенный метод (функция-член) обрабатывал данные, встроенные в объект, мы активируем функцию-член для объекта:

```
object.member_function()
```

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string secret = "abracadabra";
    string password;
    cout << "Enter password:" << endl;
    getline(cin, password);
    if (secret == password)
        cout << "Access granted" << endl;
    else
        cout << "Sorry";
}
Enter password:
mama
Sorry
```

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string secret = "abracadabra";
    string password;
    cout << "Enter password:" << endl;
    getline(cin, password);
    if (secret.compare(password) == 0)
        cout << "Access granted" << endl;
    else
        cout << "Sorry";
}
Enter password:
abcde
Sorry
```

Заменили вхождение оператора == эквивалентной активацией функции-члена сравнения.

Выражение: secret.compare(password)

В нем говорится: активируйте функцию-член сравнения compare для secret объекта, чтобы сравнить строку, хранящуюся в secret, со строкой, хранящейся в password.

Можно записать это выражение password.compare(secret) без изменения поведения программы.

Функции-члена compare может диагностировать все возможные отношения между двумя строками:

```
str1.compare(str2) == 0 если  str1 == str2

str1.compare(str2) > 0 если  str1 > str2

str1.compare(str2) < 0  если  str1 < str2
```

Подстрока – часть строки.

Функция-член substr:

```
    newstr = oldstr.substr(substring_start_position,
length_of_substring)
```

Подстрока любой строки определяется:

• местом, где начинается подстрока (параметр substring_start_position);

• его длиной (параметр length_of_substring).

Оба параметра имеют значения по умолчанию.

- s.substr(1,2) описывает подстроку строки s, начинающуюся со 2-го символа и заканчивающуюся 3-им символом (включительно)

- s.substr(1) описывает подстроку, начинающуюся со 2 символа строки s и содержащую все оставшиеся символы s, включая последний; опущенный параметр length_of_substring по умолчанию охватывает все оставшиеся символы в строке s

s.substr() - это просто копия всей строки (параметры substring_start_position по умолчанию равны 0) s.substr() - это просто копия всей строки

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string str1 = "ABCDEF";
    string str2 = str1.substr(1, 1) + str1.substr(4) + str1.substr();

    cout << str2 << endl;
}
```

Результат:

BEFABCDEF

Каждая строка имеет определенную длину. Даже пустая строка имеет нулевую длину. Информация о длине строки предоставляется двумя функциями-членами. Их имена разные, но их поведение идентично. Эти функции являются синонимами.

Прототипы:

```
int string_size = any_string.size();

int string_length = any_string.length();
```

Обе эти переменные int равны.

Обе функции возвращают значение, равное количеству всех символов, которые в данный момент хранятся в строке.

Функция substr имеет свои собственные функции-члены: substr() или size().

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string str = "12345";
  int pos = 1;
  cout << str.substr(pos).substr(pos).substr(pos).size() << endl;
}
```

Запустите программу со строкой int pos = 1; измените на int pos = 2;

Результат?

Запустите программу со строкой int pos = 1; измените на int pos = 2;

Результат?

Для нахождения подстроки в другой строке (можно искать подстроку или один символ) нужно использовать один из вариантов функции-члена find .

Два из них особенно полезны:

```
int where_it_begins = any_string.find(another_string, start_here);
```

```
int where_it_is = any_string.find(any_character, start_here);
```

Параметр start_here по умолчанию равен 0, если его опустить, поиск строки будет производиться с самого начала.

Результат, возвращаемый функциями, указывает на первое место в строке, где начинается искомая строка или где находится искомый символ.

Если поиск завершается неудачно, обе функции возвращают специальное значение string::npos. Его можно использовать, чтобы проверить, содержит ли строка нужную подстроку.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string greeting = "My name is Bond, James Bond.";
  string weneedhim = "James";
  if (greeting.find(weneedhim) != string::npos)
    cout << "OMG! He's here!" << endl;
  else
    cout << "It's not him." << endl;

  int comma = greeting.find(',');
  if (comma != string::npos)
    cout << "Interesting. He used a comma." << endl;
}
OMG! He's here!
Interesting. He used a comma.
```

Результат:

OMG! He's here!

Interesting. He used a comma.

Можно управлять размером памяти, используемой строкой, с помощью функции-члена reserve(). Он может работать в обоих направлениях: сжимать и расширять буферы.

Для функции требуется один параметр типа int, чтобы указать желаемый размер выделенных буферов.

Важно: содержимое строки не изменяется (содержимое невосприимчиво к действию функции reserve()).

При определении конечного размера выделенной памяти функция может округлять значение параметра, чтобы соответствовать текущим требованиям к памяти и/или условиям целевой платформы.

```
#include <iostream>

#include <string>

using namespace std;

void print_info(string& s)

{

  cout << "content =\"" << s << "\" ";

  cout << "capacity = " << s.capacity() << endl;

  cout << "---------" << endl;

}

int main()

{

  string the_string = "content";

  print_info(the_string);

  the_string.reserve(100);

  print_info(the_string);

  the_string.reserve(0);

  print_info(the_string);

}
```

Результат:

content ="content" capacity = 15

---

content ="content" capacity = 100

---

content ="content" capacity = 15

Содержимое строки остается нетронутым в каждом случае.

```
#include <iostream>
#include <string>
using namespace std;
void print_info(string& s)
{
  cout << "content =\"" << s << "\" ";
  cout << "capacity = " << s.capacity() << endl;
  cout << "is empty? " << (s.empty() ? "yes" : "no") << endl;
  cout << "---------" << endl;
}
int main()
{
  string the_string = "content";
  print_info(the_string);
  the_string.resize(50, '?');
```

```
  print_info(the_string);
  the_string.resize(4);
  print_info(the_string);
  the_string.clear();
  print_info(the_string);
}
```

Результат:

content ="content" capacity = 15

is empty? no

---

content ="content??????????????????????????????????????" capacity = 50

is empty? no

---

content ="cont" capacity = 50

is empty? no

---

content ="" capacity = 50

is empty? yes

Можно получить доступ к каждому символу отдельно. Это нужно во многих алгоритмах (шифрование текста - нужно изменять каждый отдельный символ char строки string).

Строки предлагают удобные функции-члены для этого.

Строки не являются векторами, но они способны представлять свое содержимое так, как если бы это был реальный вектор. Это позволяет читать и писать каждый символ отдельно.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string the string = "content";
    for (int i = 0; i < the string.length(); i++)
        the string[i] = the string[i] - 'a' + 'A';
    cout << the string << endl;
}
CONTENT
```

Программа - простая, преобразует каждый символ в верхний регистр.

Как строка может сделать это? - Это связано с механизмом “перегрузка оператора”.

Функция append() - для добавления одной строки к другой.

Эту же задачу может выполнять оператор +=.

Функция append() предоставляет больше возможностей, более гибкая, чем +=.

Добавление строки :

```
    string str1 = "content"; str2 = "appendix";

str1.append(str2);

        // str1 содержит "contentappendix" сейчас
```

Добавление не только строки но и подстроки строки:

```
    string str1 = "content";  str1.append("tail",1,3);
```

```
// str1 содержит "contentail" сейчас
```

Добавление символа (возможно, повторяющегося n раз):

```
string str1 = "content";  str1.append(3, 'x');

// str1 содержит "contentxxx" сейчас
```

Эти варианты влияют на содержимое строки и возвращают измененную строку в результате. Можно использовать этот эффект, например, создавая цепочку вызовов append() .

```
#include <iostream>

#include <string>

using namespace std;

int main()

{

  string the_string = "content";

  string new_string;

  new_string.append(the_string);

  new_string.append(the_string, 0, 3);

  new_string.append(2, '!');

  cout << new_string << endl;

}
```

Результат:

contentcon!!

Если нужно добавить в строку только один символ, это можно сделать это с помощью функции append.

Есть более эффективный способ - с помощью функции-члена push_back

.

Пример - классический латинский алфавит.

```
#include <iostream>

#include <string>

using namespace std;

int main (void) {

  string the_string;

  for (char c = 'A'; c <= 'Z'; c++)

    the_string.push_back(c);

  cout << the_string << endl;

  return 0;

}
```

Результат:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

Вставка строки в строку подобна расширению ее содержимого изнутри. Новый набор содержимого просто “вставляется” “pushed” в старый.

Фрагмент кода выведет “быть или не быть” в поток cout:

```
string quote = "to be ";

quote.append(quote);

quote.insert(6, "or not ");

cout << quote << endl;
```

Первый параметр указывает, где должна быть вставка, второй - что там должно быть вставлено.

Это только одна из возможностей, предлагаемых функцией.

Другая функция-член может вставлять подстроку указанной строки способом, похожим на функцию добавления.

Функция, которая может вставлять один символ, при необходимости дублируемый более одного раза. Обе эти функции используются в следующем примере

```
#include <iostream>

#include <string>

using namespace std;

int main()

{

  string quote = "Whyserious?", anyword = "monsoon";

  quote.insert(3, 2, ' ').insert(4, anyword, 3, 2);

  cout << quote << endl;

}
```

Результат:

Why so serious?

Функция-член assign() похожа на вставку, но не сохраняет предыдущее содержимое строки, а просто заменяет его новым.

Зачем использовать функцию assign(), когда эту же задачу может выполнить оператор =.

Функция assign() так же универсальна, как функции insert() или append(), поэтому более удобна в некоторых конкретных приложениях.

```
#include <iostream>

#include <string>

using namespace std;

int main()

{

  string sky;

  sky.assign(80, '*');

  cout << sky << endl;

}
```

Результат:

****
****

Функция-член replace() - заменяет часть строки другой строкой или подстрокой другой строки.

Для указания, какую часть строки нужно заменить, нужно указать два числа:

первое - описание начальной позиции;

второе - сколько символов будет заменено.

Первой перегруженной функции требуется строка (в качестве третьего параметра) для замены старого содержимого (оно может быть либо длиннее, либо короче, либо равно по размеру по сравнению с оригиналом).

Вторая функция позволяет указать подстроку, которая будет использоваться в качестве подстановки.

Второй вариант функции- в примере.

Функция не только изменяет строку, но и возвращает измененную строку в результате Это можно использовать для построения серии замен, происходящих в одном операторе.

```
#include <iostream>

#include <string>

using namespace std;

int main()

{

  string to_do = "I'll think about that in one hour";

  string schedule = "today yesterday tomorrow";

  to_do.replace(22, 12, schedule, 16, 8);

  cout << to_do << endl;

}
```

Результат:

I'll think about that tomorrow

Функция-член erase() - удаляет часть строки, делает ее короче.

Для выполнения функции требуется два числа:

- место, где начинается удаляемая подстрока (по умолчанию это значение = 0);

- длина подстроки (значение по умолчанию = длине исходной строки).

Вызов: the_string.erase();

удаляет все символы из строки и оставляет ее пустой.

Соглашение, которое заставляет функцию вносить изменения и возвращать измененную строку в результате, все еще действует, поэтому пример выведет:

I've got a feeling we're in Kansas

Последовательность вызовов стирания не может быть отменена, утверждение:

where_are_we.erase(25, 4).erase(38, 8);

приведет к совершенно другим результатам.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string where_are_we = "I've got a feeling we're not in Kansas anymore";
  where_are_we.erase(38, 8).erase(25, 4);
  cout << where_are_we << endl;
}
```

Результат:

I've got a feeling we're in Kansas

Функция-член swap() – обмен содержимым двух строк.

Эта функция во много раз быстрее, чем обычная (физически выполняемая) замена.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string drink = "A martini";
  string needs = "Shaken, not stirred";
  cout << drink << ". " << needs << "." << endl;
  drink.swap(needs);
  cout << drink << ". " << needs << "." << endl;
}
```

Результат:

A martini. Shaken, not stirred. Shaken, not stirred. A martini.

Пространства имен

Пространство имен - это пространство, в котором конкретное имя имеет однозначное и ясное значение.

Без using namespace std;

ошибка

```
#include <iostream>

int main()

{

  cout << "Play it, Sam" << endl;

}
```

Можно:

```
#include <iostream>

int main()

{

  std::cout << "Play as time goes by" << std::endl;

}
```

Play as time goes by

Оператор :: -оператор разрешения области действия

Определение пространства имен:

```
namespace the_name_of_the_space  {

}
```

Любая сущность, объявленная внутри пространства имен (между { }), будет привязана к этому пространству имен и логически отделена от любой другой сущности с тем же именем.

```
#include <iostream>

using namespace std;

namespace Hogwarts {

  int troll = 1;

}

namespace Mordor {

  int troll = 2;

}

int main()

{

  cout << Hogwarts::troll << " " << Mordor::troll << endl;

}
```

Результат:

1 2

Если какое-либо из доступных пространств имен является более удобным или предпочтительным, его можно использовать так, чтобы предполагалось, что компилятор пытается квалифицировать каждое неквалифицированное имя с помощью этого/этих имен пространства имен (может быть более одного пространства имен этого типа).

Действие по использованию выбранного пространства имен выполняется оператором using namespace. Если оператор помещен вне какого-

либо блока (часть кода в скобках { }), это влияет на код после оператора до конца исходного файла.

Пример - случай, когда в одном и том же коде указаны два оператора using namespace.

Инструкции namespace не должны приводить к ситуации, когда идентификатор может считаться исходящим из более чем одного пространства имен.

Не разрешается использовать следующие два оператора в одной и той же области кода:

```
using namespace Hogwarts;

using namespace Mordor;

#include <iostream>

using namespace std;

namespace Hogwarts {

  int troll = 1;

}

namespace Mordor {

  int troll = 2;

}

using namespace Hogwarts;

int main()

{

  cout << troll << " " << Mordor::troll << endl;
```

```
}
```

Результат:

1 2

Если оператор using namespace помещен внутри блока, его область действия заканчивается в том же месте, где заканчивается блок. Можно использовать этот эффект для выборочного использования (и неиспользования) любого из доступных пространств имен. В примере использованы все три доступных пространства имен.

Пространство имен std используется глобально (во всем исходном файле). Пространства имен Hogwarts и Mordor используются выборочно.

```
#include <iostream>
using namespace std;
namespace Hogwarts {
  int troll = 1;
}
namespace Mordor {
  int troll = 2;
}
int main()
{
  {
    using namespace Hogwarts;
    cout << troll << " ";
```

```
  }
  {
    using namespace Mordor;
    cout << troll << endl;
  }
}
```

Результат:

1 2

В примере показано, как расширить любое ранее определенное пространство имен. Код содержит “двойные” определения пространств имен. На самом деле они не удваиваются – они расширяются.

Каждое расширение может быть помещено в отдельный исходный файл. Это означает, что любое из пространств имен может быть распределено между множеством исходных файлов, созданных разными разработчиками.

Первое появление пространства имен называется “исходным пространством имен”.

Любое пространство имен с тем же именем, которое появляется после исходного пространства имен, называется “пространством имен расширения”.

В пространстве имен может быть несколько расширений.

```
#include <iostream>
using namespace std;


namespace Hogwarts{
```

```
    int troll = 1;
}
namespace Mordor {
    int troll = 2;
}
namespace Hogwarts {
    float wizard = -0.5;
}
namespace Mordor {
    float wizard = 0.5;
}
int main()
{
cout << Hogwarts::troll << " " << Hogwarts::wizard << endl;
cout << Mordor::troll << " " << Mordor::wizard << endl;
}
```

Результат:

1 -0.5

2 0.5

Объявление using использует всё пространство имен, неявно определяет все сущности, привязанные к этому пространству. Существует форма утверждения, которая позволяет выборочно решать, какие объекты следует использовать, а какие должны оставаться скрытыми внутри пространства.

Важное условие: ни один из используемых операторов не может вызвать двусмысленность в процессе идентификации всех сущностей, используемых в коде.

Пример - используется оператор using для выбора двух разных существ из двух разных миров.

```
#include <iostream>

using namespace std;

namespace Hogwarts {

    int Troll = 1;

    float Wizard = -0.5;

}

namespace Mordor {

    int Troll = 2 ;

    float Wizard = 0.5;

}

using Mordor::Troll;

using Hogwarts::Wizard;

int main()

{

    cout << Hogwarts::Troll << " " << Wizard << endl;

    cout << Troll << " " << Mordor::Wizard << endl;

}
```

Результат:

1 -0.5

2 0.5

Можем определить пространство имен без имени (анонимное пространство имен).

Этот тип пространства имен неявно и автоматически используется в исходном файле, где видно его определение. Это еще один способ представления сущностей (например, переменных), доступных через весь исходный файл.

Пример показывает два пространства имен: анонимное и созданное с именем

```
#include <iostream>

using namespace std;

namespace {

  int troll = 1;

  float wizard = -0.5;

}

namespace Mordor {

  int troll = 2;

  float wizard = 0.5;

}

int main()

{

  cout << troll << " " << wizard << endl;
```

```
  cout << Mordor::troll << " " << Mordor::wizard << endl;
}
```

Результат:

1 -0.52 0.5

Можно переименовать пространство имен локально, он будет распознан под новым именем только в исходном файле, в котором произошло переименование.

Можно использовать его со специальной формой инструкции пространства имен:

```
namespace new_name = old_name;
```

Новое имя пространства имен может использоваться вместе со старым, оба имени действительны.

Пример показывает, как длинное и запутанное имя может быть изменено, чтобы облегчить жизнь разработчику.

```
#include <iostream>
using namespace std;
namespace what_a_wonderful_place_for_a_young_sorcerer {
    int troll = 1;
    float wizard = -0.5;
}
namespace Mordor {
    int troll = 2;
```

```
        float wizard = 0.5;
    }
    namespace                    Hogwarts                    =
what_a_wonderful_place_for_a_young_sorcerer;
    int main()
    {
         cout << Hogwarts::troll << " " <<
         what_a_wonderful_place_for_a_young_sorcerer::wizard  <<
endl;
         cout << Mordor::troll << " " << Mordor::wizard << endl;
    }
```

Результат:

1 -0.52 0.5