**27 марта 2020г Лекция по дисциплине «Программирование»**

## АРИФМЕТИКА УКАЗАТЕЛЕЙ (продолжение предыдущей лекции)

### ③Сравнение указателей

Ещё одной типовой операцией для работы со стеком, является операция проверки его пустоты.

Теоретические сведения по теме «Стек», в том числе по проверке его пустоты, смотри в видео-материале курса «Алгоритмы программирования и структуры данных» от университета ИТМО (время с 00:00 по 05:30). Источник: https://openedu.ru.

**Упражнение по программированию.**

После изучения видео предлагается изменить программу по работе со стеком (см. предыдущую лекцию):

- добавить в неё новую функцию `bool Empty();` , которая возвращает `true`, если стек пуст,
- заменить имеющиеся в программе проверки на пустоту стека вызовом функции `Empty()`.

**Промежуточный контроль знаний**
**(обязательно для выполнения всеми)**

Доработанную программу по работе со стеком необходимо выслать на почту преподавателя в виде исходного кода. В теме письма указать *ФИО, группу, стек*. Например:

*Иванов ИИ, ИВТ-41-19, стек*

Срок выполнения: 30 марта. Вовремя присланные работающие без ошибок программы повышают экзаменационную оценку в конце семестра на 0,5 балла (при условии выполнения остальных работ до начала зачётной недели).

**Дополнительное упражнение по программированию**
**(выполняется по желанию)**

Ещё одной типовой структурой данных, используемой в программировании, является *очередь*. Предлагается разработать программу, которая демонстрирует работу с очередью, по аналогии с программой по работе со стеком.

Теоретические сведения по теме «Очередь» смотри в видео-материале курса на «Алгоритмы программирования и структуры данных» от университета ИТМО (время с 05:30). Источник: https://openedu.ru.

Для представления очереди необходимо использовать массив с максимальным количеством элементов 20. Реализовать операции добавления и удаления элемента очереди, вывод содержимого очереди, проверка на пустоту, проверка на полноту очереди. Общение с пользователем реализовать в режиме диалога. Строки программа должны быть снабжены содержательными комментариями.

Программа высылается на почту преподавателя в виде исходного кода. В теме письма указать *ФИО, группу, очередь*. Например:

*Иванов ИИ, ИВТ-41-19, очередь*

Срок выполнения: 01 апреля. Вовремя присланные работающие без ошибок программы повышают экзаменационную оценку в конце семестра ещё на 0,5 балла.

### *Дополнение к массивами*

Как уже было сказано, *имя массива* интерпретируется как *адрес первого элемента массива*. Однако из этого правила есть исключения.

### Адрес массива

Получение адреса массива является случаем, при котором имя массива не интерпретируется как его адрес. Применение операции взятия адреса приводит к выдаче адреса целого массива:

```
int a[2];
cout << a ;  // отображение &a[0] – адреса первого элемента массива
cout << &a;  // отображение адреса целого массива
//  0x23fdf0  0x23fdf0
```

С точки зрения числового представления эти два адреса одинаковы, но концептуально `&a[0]` и, следовательно, `a` — это адрес 4-байтного блока памяти, тогда как `&a` — адрес 8-байтного блока памяти. Таким образом, выражение `a+1` приводит к добавлению 4 к значению адреса, а `&a+1` — к добавлению 8 к значению адреса:

```
cout << a+1 << " " << &a+1;
//  0x23fdf4  0x23fdf8
```

Эту ситуацию можно сформулировать и по-другому: `a` имеет тип "указатель на int", или `int *`, а `&a` имеет тип "указатель на массив из 2х элементов int", или `int(*)[2]` (рис. 1).

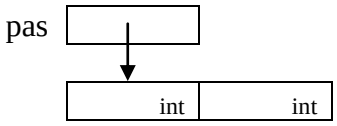


Рис. 1. Указатель на массив из двух целочисленных элементов.

Теперь вас может заинтересовать происхождение последнего описания типа. Сначала посмотрим, как можно объявить и инициализировать указатель этого типа:

```
int (*pas) [2] = &a; // pas указывает на массив из 2х элементов int
```

Если опустить круглые скобки, то правила приоритетов будут ассоциировать `[2]` в первую очередь с `pas`, делая `pas` массивом из 2х указателей на `int` (рис. 2), поэтому круглые скобки необходимы. Тогда описание типа переменной (без имени переменной) `pas` будет:

```
int (*) [2]    .
```

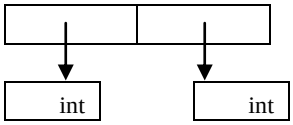


Рис. 2. Массив указателей на целочисленные динамические значения.

Кроме того, обратите внимание, что поскольку значение `pas` установлено в `&a`, `*pas` эквивалентно `a`, и `(*pas)[0]` будет первым элементом массива `a`.

**Вопросы для самостоятельного изучения:**

а). Какой смысл выражения `*pas[0]`?

### Размер массива

Другим исключением из общего правила является операция `sizeof`. Повторение известного:

```
// указатели не имеют полной информации о массиве
int * b = new int [2];  // динамический массив
cout << sizeof b << " " << sizeof b[0];
// 8  4   -  размер указателя 8 байт (зависит от операционной системы)
// НО!!!
int a[5]; // статический массив
cout << sizeof a << " " << sizeof a[0];
// 20  4   - размер всего массива и его одного элемента
```

*Выводы* из примера:

– Применение `sizeof` к имени статического массива возвращает размер массива в байтах, но применение `sizeof` к указателю возвращает размер указателя, даже если он указывает на массив.

Говоря кратко, использовать `new` для создания массива и применять указатели для доступа к его различным элементам очень просто. Вы просто трактуете указатель как имя массива. Однако понять, почему это работает — интересная задача. Если вы действительно хотите понимать массивы и указатели, то должны тщательно исследовать их поведение, связанное с изменчивостью.

## УКАЗАТЕЛИ И СТРОКИ

Специальные отношения между массивами и указателями расширяют строки в стиле С. Рассмотрим следующий код:

```
char flower[10] = "rose";
cout << flower << "s are red\n";
```

Имя массива — это адрес его первого элемента, потому `flower` в операторе `cout` представляет адрес элемента `char`, содержащего символ `'r'`. Объект cout предполагает, что адрес `char` — это адрес строки, поэтому печатает символ, расположенный по этому адресу, и затем продолжает печать последующих символов до тех пор, пока не встретит нулевой символ (\0). Короче говоря, вы сообщаете `cout` адрес символа, он печатает все, что находится в памяти, начиная с этого символа и до нулевого.

Ключевым фактором здесь является не то, что `flower` — имя массива, а то, что `flower` трактуется как адрес значения `char`. Это предполагает, что вы можете использовать переменную-указатель на `char` в качестве аргумента для `cout`, потому что она тоже содержит адрес `char`. Разумеется, этот указатель должен указывать на начало строки. Чуть позже мы проверим это.

Но как насчет финальной части предыдущего оператора `cout`? Если `flower` — на самом деле адрес первого символа строки, что собой представляет выражение `"s are red\n"`? Согласно принципам, которыми руководствуется `cout` при выводе строк, эта строка в кавычках также должна быть адресом. Так оно и есть: в C++ строка в кавычках, как и имя массива, служит адресом его первого элемента. Предыдущий код на самом деле не посылает полную строку объекту `cout`; он просто передает адрес строки. Это значит, что строки в массиве, строковые константы в кавычках и строки, описываемые указателями — все обрабатываются одинаково. Каждая из них передается в виде адреса. Конечно, это уменьшает объем работ компьютеру по сравнению с тем, который потребовался бы в случае передачи каждого символа строки.

*!!! С объектом cout, как и в большинстве других выражений C++, имя массива `char`, указатель на `char`, а также строковая константа в кавычках — все интерпретируются как адрес первого символа строки.*

В следующем примере иллюстрируется применение различных форм строк. Код в этом листинге использует две функции из библиотеки обработки строк. Функция `strlen()`, которую вы уже применяли ранее, возвращает длину строки. Функция `strcpy()` копирует строку из одного места в другое. Обе функции имеют прототипы в файле заголовков `cstring` (или `string.h` — в устаревших реализациях). В программе также присутствуют комментарии, предупреждающие о возможных случаях неправильного применения указателей, которых следует избегать.

***Пример.***

```
// ptrstr.cpp - использование указателей на строки
#include <iostream>
#include <cstring> // объявление strlen(), strcpy()
int main()
{
    using namespace std;
    char animal[20] = "bear"; // animal содержит bear
    const char * bird = "wren"; // bird содержит адрес строки
    char * ps; //не инициализировано
    cout << animal << " and "; // отображение bear
    cout << bird << "\n"; // отображение wren
    // cout << ps << "\n"; // может отобразить мусор, но может вызвать
    // и аварийное завершение программы
    cout << "Название животного: ";
    cin >> animal; // нормально, если вводится меньше 20 символов
    // cin >> ps; очень опасная ошибка, чтобы пробовать;
    // ps не указывает на выделенное пространство
    ps = animal; // установка ps в указатель на строку
    cout << ps << "!\n"; // нормально; то же, что и применение animal
    cout << "До использования strcpy():\n";
    cout << animal << " at " << (int *) animal << endl;
    cout << ps << " at " << (int *) ps << endl;
    ps = new char[strlen(animal) +1]; // получение нового хранилища
    strcpy(ps, animal); // копирование строки в новое хранилище
    cout << "После использования strcpy():\n";
    cout << animal << " at " << (int *) animal << endl;
    cout << ps << " at " << (int *) ps << endl;
```

```
    delete [] ps;
    return 0;
}
```

**Результаты** работы программы:

```
bear and wren
Название животного: кот
кот!
До использования strcpy():
кот at 0x23fdc0
кот at 0x23fdc0
После использования strcpy():
кот at 0x23fdc0
кот at 0x6a1530
```

**Пояснения** к примеру:

①Программа создает один массив `char` (`animal`) и две переменных типа "указатель на `char`" (`bird` и `ps`). Программа начинается с инициализации массива `animal` строкой `"bear"` — точно так же, как вы инициализировали массивы и раньше. Затем программа делает нечто новое. Она инициализирует указатель на `char` строкой:

```
const char * bird = "wren"; // bird содержит адрес строки
```

Вспомните, что `"wren"` на самом деле представляет собой адрес строки, поэтому приведенный выше оператор, присваивает адрес `"wren"` указателю `bird`. (Обычно компилятор выделяет область памяти для размещения строк в кавычках, указанных в исходном коде, ассоциируя каждую сохраненную строку с ее адресом.) Это значит, что указатель `bird` можно применять так же, как использовалась бы строка `"wren"`, например:

```
cout << bird << " - птица\n";
```

Строковые литералы являются константами; именно поэтому в объявлении присутствует ключевое слово `const`. Применение `const`, таким образом, означает, что вы можете использовать `bird` для доступа к строке, но не можете изменять ее. И, наконец, указатель `ps` остается неинициализированным, поэтому он не может указывать ни на какую строку (Как вы знаете, это плохая идея, и данный пример — не исключение.)

②Далее программа иллюстрирует тот факт, что имя массива `animal` и указатель `bird` можно использовать с `cout` совершенно одинаково. Оба они являются адресами строк, и `cout` отображает две строки (`"bear"` и `"wren"`), расположенные по указанным адресам.

③Если вы удалите знаки комментария с кода, который пытается отобразить `ps`, то можете получить пустую строку, какой-нибудь мусор или даже привести программу к аварийному завершению.

④Что касается ввода, то здесь ситуация несколько отличается. Использовать массив `animal` для ввода безопасно до тех пор, пока ввод достаточно краток, чтобы уместиться в отведенный массив. Однако было бы неправильно применять для ввода указатель `bird`.

• Некоторые компиляторы трактуют строковые литералы как константы, доступные только для чтения, что ведёт к ошибкам времени выполнения при попытках записи в них данных. То, что строковые литералы являются константами — обязательное поведение C++, однако пока не все разработчики компиляторов отказались от старого подхода при их обработке.

• Некоторые компиляторы используют только одну копию строкового литерала для представления всех его вхождений в тексте программы.

⑤Давайте проясним второй пункт. C++ не гарантирует уникальное сохранение строкового литерала. То есть, если вы используете литерал `"wren"` несколько раз в программе, компилятор может сохранить либо несколько копий этой строки, либо всего одну. Если он сохраняет одну, то установка в `bird` адреса `"wren"` заставляет его указать на единственную копию этой строки. Чтение нового значения в эту одну строку может повлиять на строки, которые изначально имели то же самое значение, но логически были совершенно независимыми, встречаясь в других местах программы.

В любом случае, поскольку указатель `bird` объявлен как `const`, компилятор предотвратит любые попытки изменить содержимое, расположенное по адресу, на который указывает `bird`.

⑥Еще хуже обстоят дела с попытками чтения информации в место, на которое указывает `ps`. Поскольку указатель `ps` не инициализирован, вы не можете знать, куда попадет введенная

информация. Она может даже перезаписать ту информацию, которая уже имеется в памяти. К счастью, такой проблемы легко избежать: вы просто применяете достаточно большой массив `char`, чтобы принять ввод, но не используете для ввода строковых констант и неинициализированных указателей (или же можете обойти все эти проблемы и работать вместо массивов с объектами `std::string`).

*!!! При вводе строки внутри программы всегда необходимо использовать адрес ранее распределенной памяти. Этот адрес может иметь форму имени массива либо указателя, инициализированного с помощью операции `new`.*

⑦Далее обратите внимание на то, что делает следующий код:

```
ps = animal; // установить в ps указатель на строку
cout << animal << " at " << (int *) animal << endl;
cout << ps << " at " << (int *) ps << endl;
```

Он генерирует следующий вывод:

```
кот at 0x23fdc0
кот at 0x6a1530
```

Обычно если объекту `cout` передается указатель, он печатает адрес. Но если указатель имеет тип `char *`, то cout отображает строку, на которую установлен указатель. Если вы хотите увидеть адрес строки, для этого потребуется выполнить приведение типа к указателю на другой тип, такой как `int *`, что и делает показанный код. Поэтому `ps` отображается как строка "`кот`", но `(int *) ps` выводится как адрес, по которому эта строка находится.

Обратите внимание, что присваивание `animal` переменной `ps` не копирует строку; оно копирует только адрес. В результате два указателя (`animal` и `ps`) указывают на одно и то же место в памяти — т.е. на одну строку.

⑧Чтобы получить копию строки, потребуется сделать кое-что еще. Первый подход предусматривает распределение памяти для хранения копии строки. Это можно сделать либо за счет объявления еще одного массива, либо с помощью операции `new`.

Второй подход позволяет точно настроить размер хранилища для строки:

```
ps = new char[strlen(animal) + 1]; // получить новое хранилище
```

Строка "`кот`" не полностью заполняет массив `animal`, поэтому при таком подходе память расходуется непроизводительно. Здесь же мы видим использование `strlen()` для нахождения длины строки, а затем к найденной длине прибавляется единица, чтобы получить длину, включающую нулевой символ. Далее программа применяет `new` для выделения достаточного пространства под хранение строки.

Вам необходим способ копирования строки из массива `animal` во вновь выделенное пространство. Установка `ps` в `animal` не работает, потому что это только изменяет адрес, сохраненный в `ps`, при этом утрачивается единственная возможность доступа к выделенной памяти. Поэтому взамен необходимо применять библиотечную функцию `strcpy()`:

```
strcpy(ps, animal); // скопировать строку в новое хранилище
```

Функция `strcpy()` принимает два аргумента. Первый представляет собой целевой адрес, а второй — адрес строки, которую следует скопировать. Ваша обязанность — обеспечить, чтобы место назначения действительно смогло вместить копируемую строку. Здесь это достигается использованием функции `strlen()` для определения корректного размера и применением операции `new` для получения свободной памяти. Обратите внимание, что за счет использования `strlen()` и `new` получены две отдельных копии "`кот`":

```
кот at 0x23fdc0
кот at 0x6a1530
```

Также обратите внимание, что новое хранилище располагается в памяти довольно далеко от того места, где хранится содержимое массива `animal`.

⑨Вам часто придется сталкиваться с необходимостью размещения строки в массиве. Для этого можно воспользоваться операцией = при инициализации массива; иначе придется иметь дело с функцией `strcpy()` или `strncpy()`. Вы уже видели функцию `strcpy()`; она работает следующим образом:

```
char food[20] = "carrots"; // инициализация
strcpy(food, "flan"); // альтернатива
```

Обратите внимание, что следующий подход может послужить причиной проблем, если массив `food` окажется меньше, чем строка:

```
strcpy(food, "a picnic basket filled with many goodies");
```

В этом случае функция копирует остаток строки в байты памяти, непосредственно следующие за массивом, при этом перезаписывая ее содержимое, несмотря на то, что, возможно, программа ее использует для других целей. Чтобы избежать такой проблемы, вместо `strcpy()` вы должны применять `strncpy()`. Эта функция принимает третий аргумент — максимальное количество копируемых символов. Однако при этом имейте в виду, что если данная функция исчерпает свободное пространство еще до достижения конца строки, то нулевой символ она не добавит. Потому применять ее нужно так:

```
strncpy(food, "a picnic basket filled with many goodies", 19) ;
food[19] = ' \0';
```

Этот код копирует до 19 символов в массив, после чего устанавливает последний элемент массива в нулевой символ. Если строка короче, чем 19 символов, то `strncpy()` добавит нулевой символ ранее, пометив им действительный конец строки.

*!!! Для копирования строки в массив применяйте `strcpy()` или `strncpy()`, а не операцию присваивания.*