1. Структуры данных и способы их реализации

Стек (stack) – это последовательность **однотипных** элементов, характерная тем, что элементы добавляются и удаляются **только с одного конца**, называемого **вершиной** стека. Стек работает по принципу «Элемент, помещенный в стек **последним**, извлечен будет **первым**». Иногда этот принцип обозначается сокращением **LIFO** (от английского «Last In – First Out», т.е. «последним зашел – первым вышел»). Естественно, можно перевернуть этот принцип и сформулировать его так: «Элемент, помещенный в стек **первым**, извлечен будет **последним**». Иногда такой элемент называют **дном** стека.

Бытовой пример стека – стопка тяжелых предметов (например – бетонные плиты). **Примеры** использования стека в программировании:

- вложенные вызовы подпрограмм, в частности – рекурсивные: вызовы подпрограмм идут в одном порядке, а возвраты из них – в обратном
- стек операндов при разборе арифметических выражений со вложенными скобками

Важность стека для вычислительной техники в целом подтверждается тем, что все современные типы процессоров реализуют стековые операции на уровне базовых команд.

Стеки могут хранить любые **однотипные** данные, как простейшие (целые числа или отдельные символы), так и структурированные (строки, записи).

Очередь (Queue) – это последовательность **однотипных** элементов, характерная тем, что новые элементы в нее **добавляются** с **одного** конца, а **удаляются** с **другого.**

Очередь работает по принципу “Элемент, помещенный в очередь **первым**, извлечен будет тоже **первым**”. Иногда этот принцип обозначается сокращением **FIFO** (от английского «First In – First Out», т.е. «Первым зашел – первым вышел»). Для очереди определяется ее **начало** (первый элемент в очереди) и **конец** (последний элемент).

Очереди особенно часто используются в **системных** программах, прежде всего в многозадачных операционных системах (очередь потоков на выполнение, очередь событий, очередь заданий на печать и т.д.). Кроме того, существует множество **прикладных** задач, связанных с обслуживанием клиентов или заявок, где без подобной структуры не обойтись.

Линейный список (**List**) – это набор связанных **однотипных** элементов, в котором каждый элемент каким-то образом определяет следующий за ним элемент. В отличие от стека и очереди, добавление нового элемента возможно в **любом месте** списка, также можно удалить **любой** элемент списка. Ясно, что списковые структуры являются более **гибкими**, но и немного более **сложными** в реализации. Фактически, стеки и очереди можно считать **частными случаями** списков, в которых добавление и удаление элементов может выполняться только на концах. Списковые структуры находят широкое применение и в системных, и в прикладных задачах. Важной разновидностью списков являются **упорядоченные** списки, в которых элементы выстроены по порядку в соответствии с некоторым правилом.

Все три перечисленные структуры данных можно реализовать одним из **двух** способов:

- на основе **массива (статическая** или **непрерывная** реализация)
- на основе использования специальных **адресных** переменных и механизма **динамического** распределения памяти **(динамическая, адресная** или **связная** реализация**)**

Ни один из этих способов не является идеальным, каждый имеет свои достоинства и недостатки и отсюда – **свою область применения**. Не вдаваясь пока в тонкости реализации этих способов, дадим следующее их **сравнение** (отметив, где это имеет смысл, плюсы и минусы).

Реализация на основе **массива**:

- может использоваться, когда число элементов в структуре **известно** хотя

бы приблизительно и **мало** изменяется в процессе эксплуатации, иначе могут возникать существенные потери памяти за счет большого числа не используемых элементов массива

- имеет очень **простую** программную реализацию, сводящуюся к стандартным операциям с массивов (+)
- обеспечивает очень **высокую скорость** выполнения таких операций как **доступ** к элементам по их индексам, **поиск** при использовании упорядоченных массивов, добавление и удаление для **стеков** и **очередей** (+)
- при реализации списков может приводить к **замедлению** таких операций как добавление и удаление элементов (-)
- требует выделения **непрерывной** области памяти, что не всегда возможно (-)

Реализация на основе **адресных связей**:

- подходит для ситуации, когда число элементов может **изменяться в очень широких пределах**
- **не требует** выделения единой **непрерывной** области памяти и приводит к более **экономному** расходованию оперативной памяти (+)
- обеспечивает **быстрое** выполнение операций **добавления** и **удаления** элементов, особенно для стеков и очередей (+)
- имеет более **сложную** программную реализацию, основанную на использовании переменных адресного типа и **увеличивающую** вероятность появления **ошибок** при выполнении программы (-)
- **многократное повторение** операций динамического выделения и освобождения памяти может приводить к некоторому **замедлению** работы программы (-)

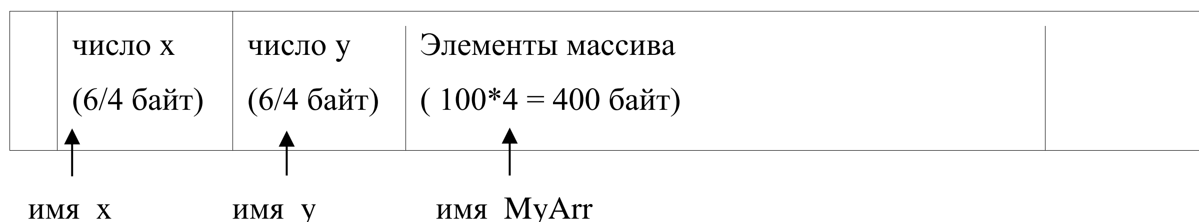
Первый способ реализации структур использует классическое понятие массива, что объясняет происхождение термина “**непрерывная**” реализация: элементы структуры занимают **последовательные** ячейки массива, т. е.

располагаются в памяти **строго друг за другом**. Второй термин (**статическая** реализация) связан с особенностями **выделения памяти** для элементов массива: это происходит при **обработке** текста программы **транслятором**, т. е. на этапе **создания** исполняемого кода.

Каждой объявленной в программе переменной транслятор выделяет **необходимую** по размеру область памяти и **связывает** адрес этой области с именем переменной. Например, пусть объявлены следующие три переменные:

var x, y : **real**; MyArr : **array** [1 .. 100] **of** **integer**;	 **float** x, y; **int** MyArr[100] ;

Тогда компилятор при обработке этих объявлений распределит память **примерно** следующим образом (с учетом возможных различий в представлении базовых типов на разных платформах):



Особенность статического распределения памяти – **жесткая** фиксация выделенных областей, **невозможность** изменения этого распределения **при выполнении** программы. Этим и объясняются как положительные, так и отрицательные стороны использования классических массивов.

В тех задачах, где подобная жесткость является слишком неудобной, можно использовать другой способ распределения памяти – **динамический**. Он позволяет выделять память под значения переменных непосредственно в процессе **выполнения** программы в ответ на **вызов** специальных стандартных **функций**. Когда необходимость в использовании этих значений исчезает, соответствующие области памяти можно **освободить** для других целей.

Это позволяет весьма экономно расходовать такой важнейший ресурс как

оперативная память, хотя и приводит к **небольшому замедлению** выполнения программы в случае **многократного** выделения памяти за счет необходимости взаимодействия с операционной системой. Это является еще одним проявлением классического **противоречия** между **скоростью** работы программы и требуемыми **объемами** памяти.

Особенности **динамической** реализации будут рассмотрены немного позже, а несколько следующих тем будут посвящены рассмотрению **первого** способа реализации структур на основе обычных **массивов.**