Тема: Абстрактный тип. АДТ. Коллекции данных.

**Абстрактный тип. АДТ**

Абстракция данных означает предоставление только важной информации внешнему миру и скрытие их фоновых данных, т. е. Представление необходимой информации в программе без предоставления деталей.

Абстракция данных - это метод программирования (и проектирования), который основан на разделении интерфейса и реализации.

Возьмем один реальный пример телевизора, который вы можете включить и выключить, изменить канал, отрегулировать громкость и добавить внешние компоненты, такие как динамики, видеомагнитофоны и проигрыватели DVD, НО вы не знаете его внутренних деталей, что вы не знаете, как он получает сигналы по воздуху или по кабелю, как он их переводит, и, наконец, отображает их на экране.

Таким образом, мы можем сказать, что телевизор четко отделяет внутреннюю реализацию от внешнего интерфейса, и вы можете играть с его интерфейсами, такими как кнопка питания, сменщик каналов и регулятор громкости, не имея нулевого знания о его внутренних компонентах.

В C ++ классы обеспечивают отличный уровень **абстракции данных** . Они обеспечивают достаточные общественные методы внешнему миру, чтобы играть с функциональностью объекта и манипулировать объектными данными, то есть состоянием, не зная, каким образом класс был реализован внутри страны.

Например, ваша программа может сделать вызов функции sort ()**,** не зная, какой алгоритм фактически использует функция для сортировки данных значений. Фактически, базовая реализация функции сортировки может меняться между версиями библиотеки, и пока интерфейс остается прежним, ваш вызов функции будет работать.

В C ++ мы используем **классы** для определения наших собственных абстрактных типов данных (ADT). Вы можете использовать объект cout класса ostream для потоковой передачи данных на стандартный вывод следующим образом:

#include <iostream>

**using** **namespace** std;

**int** main() {

cout << "Hello C++" <<endl;

**return** 0;

}

Здесь вам не нужно понимать, как cout отображает текст на экране пользователя. Вам нужно знать только открытый интерфейс, а базовая реализация «cout» может быть изменена.

## Доступ к ярлыкам для выполнения абстракции

В C ++ мы используем метки доступа для определения абстрактного интерфейса для класса. Класс может содержать ноль или более ярлыков доступа -

* Члены, определенные с помощью общедоступного ярлыка, доступны для всех частей программы. Представление абстракции данных типа определяется его публичными членами.
* Члены, определенные с помощью частного ярлыка, недоступны для кода, который использует класс. Частные разделы скрывают реализацию от кода, который использует этот тип.

Нет ограничений на то, как часто может появляться метка доступа. Каждая метка доступа указывает уровень доступа следующих определений элементов. Указанный уровень доступа остается в силе до тех пор, пока не встретится следующая метка доступа или не увидит закрытие правой скобки тела класса.

## Преимущества абстракции данных

Абстракция данных обеспечивает два важных преимущества -

* Внутренние элементы класса защищены от непреднамеренных ошибок на уровне пользователя, что может привести к повреждению состояния объекта.
* Реализация класса может развиваться со временем в ответ на изменение требований или отчетов об ошибках без изменения кода пользователя.

Определяя члены данных только в приватном разделе класса, автор класса может вносить изменения в данные. Если реализация меняется, необходимо изучить только код класса, чтобы увидеть, что может повлиять на изменение. Если данные общедоступны, то любая функция, которая напрямую обращается к элементам данных старого представления, может быть нарушена.

## Пример абстракции данных

Любая программа на C ++, где вы реализуете класс с общедоступными и частными членами, является примером абстракции данных. Рассмотрим следующий пример:

#include <iostream>

**using** **namespace** std;

**class** Adder {

**public**:

// constructor

Adder(**int** i = 0) {

total = i;

}

// interface to outside world

**void** addNum(**int** number) {

total += number;

}

// interface to outside world

**int** getTotal() {

**return** total;

};

**private**:

// hidden data from outside world

**int** total;

};

**int** main() {

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

cout << "Total " << a.getTotal() <<endl;

**return** 0;

}

Когда приведенный выше код компилируется и выполняется, он производит следующий результат:

Total 60

Класс выше добавляет числа вместе и возвращает сумму. Публичные участники - addNum и getTotal - это интерфейсы для внешнего мира, и пользователь должен знать их, чтобы использовать класс. Общее **количество пользователей** - это то, о чем пользователю не нужно знать, но необходимо, чтобы класс работал правильно.

## Стратегия проектирования

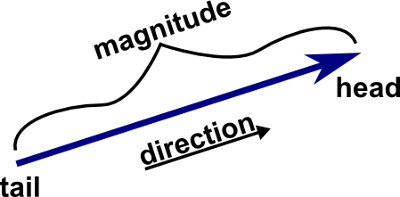
Абстракция разделяет код на интерфейс и реализацию. Поэтому при разработке вашего компонента вы должны поддерживать независимый интерфейс от реализации, чтобы при изменении базовой реализации интерфейс оставался неповрежденным.

В этом случае, независимо от того, какие программы используют эти интерфейсы, они не будут затронуты и просто нуждаются в перекомпиляции с последней реализацией.

**Коллекции данных.**

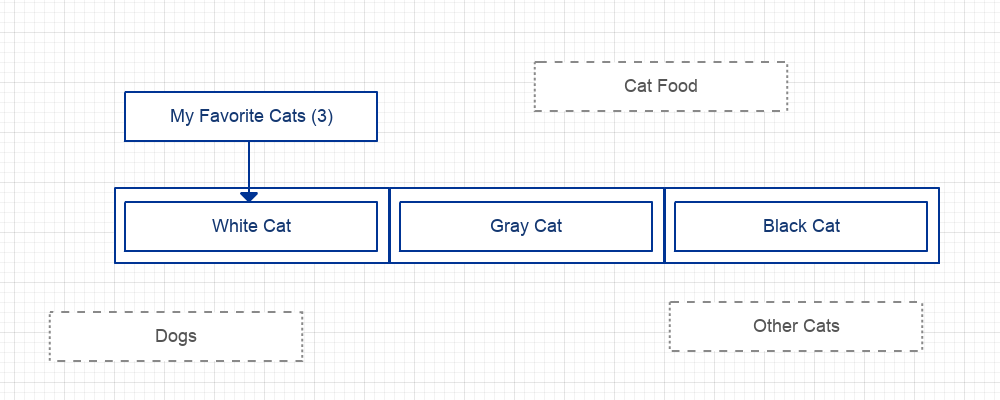
Коллекция — структура данных (тип, класс, даже лучше сказать интерфейс), которая создана, чтобы содержать в себе некоторое количество объектов (в зависимости от языка и терминологии они должны быть одного типа или могут быть разных типов).  
  
Различные типы коллекций могут быть статическими или динамическими, т.е. изменять свой размер или оставаться постоянными, могут быть упорядоченными (точнее учитывающими порядок элементов) и неупорядоченными (соответственно не учитывающими).  
  
Над коллекциями предусмотрено несколько стандартных операций (сейчас мы поговорим о мутабельных, т.е. изменяемых коллекциях), таких как: получение размера, добавление элемента, удаление элемента, поиск (есть какой-либо элемент в коллекции или нет), их очень много.

## 1 Вектор (Vector, Array)

  
  
  
Вектор (он же одномерный массив) — упорядоченный набор элементов с произвольным доступом по числовому индексу.

Доступ к элементам производится по числовому индексу (обычно начиная с 0-го индекса, хотя есть и исключения), обычно доступ к элементу коллекции по индексу записывается как myFavoriteCats[i] или blackKitties[5]. Причем для обозначения этого самого числа — индекса используют букву i.  
  
А когда одной буквы не хватает приплетают сюда j и k.  
  
Итак, далее мы понимаем, что доступ произвольный — значит мы можем обращаться к элементам под индексами 0, 42, 2014 и вобщем-то ожидаем, что операция будет сложности O(1), т.е. константной и независимо от того какой из элементов мы запросим он нам со скоростью света тут же вернется.  
  
Далее — вектор — упорядоченная коллекция, что собственно понятно — у нас есть такие понятия как первый, последний элемент, для каждого конкретно взятого элемента мы также можем назвать предыдущий и следующий.

#### **Релизация**

Обычно вектор (как низкоуровневая структура) будет представлять из себя дескриптор, содержащий различную информацию, неотделимую от самой структуры (разумнее всего держать там только размер вектора) и указатель на первый элемент.  
  
Такая реализация позволит за константное время получить доступ к произвольному элементу вектора по его индексу, а также позволит выполнять копирование, конкатенацию и другие простые операции на низком уровне.  
  
И действительно, получить доступ к определенному элементу очень просто — прибавляем к указателю на первый элемент индекс (с некоторыми поправками на размер типа данных) и получаем указатель на нужный элемент! Осталось разыменовать и у нас в переменной нужная кошечка!  
  
Ладно, вектор — классная структура, но и у него есть недостатки, например нельзя просто так взять и добавить в вектор новый элемент! Особенно втиснуть его в середину. Нельзя также сказать, что кошки с номерами 0, 1 и 4 у нас есть, а с номерами 2 и 3 — нет (раньше они были, но оказалось, что это собаки).  
  
Можно представить себе вектор, как книжную полку с отделениями, в каждом из которых помещается ровно одна книга. Чтобы засунуть новый роман Донцовой между 10-ым и 11-ым томом Большой Совецкой Энциклопедии нужно сильно постараться и переложить все тома с 11-го по 65-ый тома.  
  


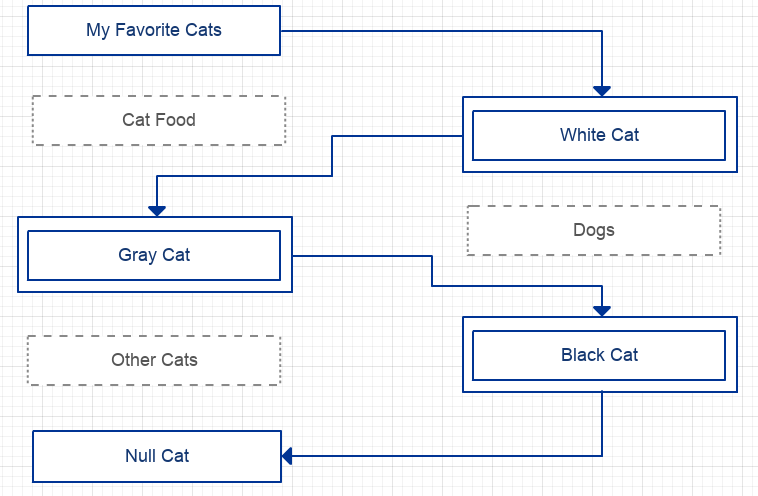
#### **Применение**

В нашем случае вектор бы идеально подошел для топ-10 самых милых котят, т.к. добавлять и удалять элементы не нужно (только изменять), пропусков между 1-ым и 5-ым местом быть не должно, да и удобно обращаться по номеру.  
  
Ладно. В любом случае вектор классный, мы просто посмотрим какие есть еще коллекции.

## 2 Список (List)

Мы уже знаем, что элементы вектора лежат акуратненько друг за другом, красиво и ровно. Это дает нам как преимущества так и недостатки.  
  
Список в этом плане полностью противоположная вещь — его элементы могут быть разбросаны по памяти как угодно! Из-за этого мы теряем возможность быстро получить элемент по индексу, а также не можем быстро скопировать весь список, но получаем довольно приятную штуку — мы можем вставлять элементы за константное время в любое место! По слухам удаляются элементы из списка тоже за O(1).

#### **Реализация**

Хм. А как с формальным определением?  
  
Список — упорядоченный набор элементов, для каждого из которых хранится указатель на следующий (или для двусвязного списка и на следующий и на предыдущий) элементы списка.  
  
Для последнего элемента списка мы храним нулевой указатель (на диаграммах я буду использовать указатель на нулевую кошку (Null Cat), не пугайтесь).  
  
Внимание! В каноничной реализации списка, для того, чтобы получить размер списка, необходимо обойти весь список — дойдя до нулевого указателя (линейное время — сложность O(n)) и хотя в некоторых реализациях размер кешируется в дескрипторе списка (или в первом элементе), не стоит на это полагаться.  
  
  
*Если бы я мог, я бы один элемент списка разместил на северном полюсе, а другой где-нибудь в окресностях Бетельгейзе*

#### **Применение**

Список бы подошел для (внимание!) списка бездомных котят, отсортированных по возрасту (по возрастанию). Нам как-раз нужно часто добавлять и удалять элементы из списка (вы не подумайте ничего такого — котят забирают), да и чаще понадобятся первые элементы списка — я бы взял себе маленького пушистого котенка, а не 8-ми-летнего манула.  
  
Ладно. Списки это вроде простая структура. Что есть еще?

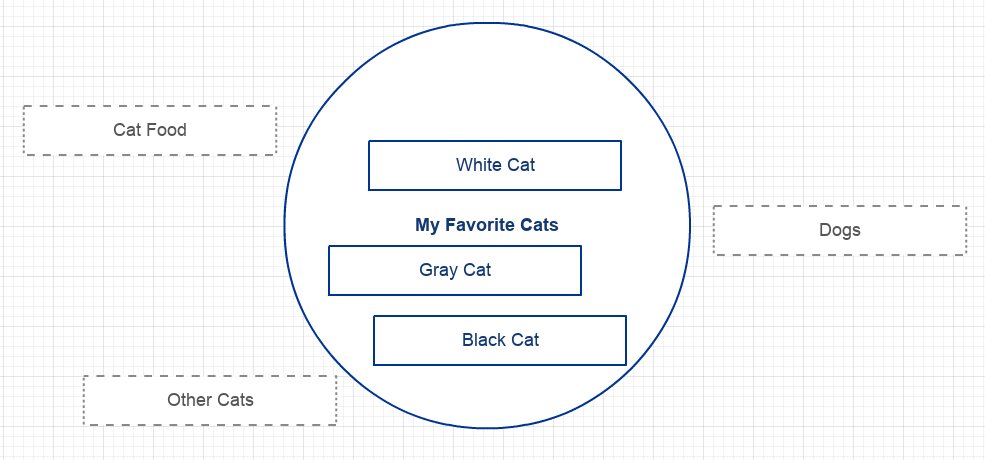
## 3 Множество (Set)

Похожее понятие есть в математике, а точнее в теории множеств. Множество отличается и от вектора и от списка, хотя их реализация может быть похожа.  
  
Множество — неупорядоченный набор элементов, без повторов.

,

Как мы знаем в векторе можно быстро получить элемент по индексу, в списке можно быстро добавить или удалить элемент, а что с множеством?  
  
В множестве можно быстро проверить, есть какой-либо элемент внутри, или его нет. Скажем если бы я хотел узнать, находится ли конкретная кошка в моем списке любимых, то и для списка и для вектора мне пришлось бы перебрать (в худжем случае) все элементы!

#### **Реализация**

В множестве, т.к. оно неупорядочено можно сортировать элементы при добавлении и в случае чего устроить бинарный поиск. Хм. Вот ведь парадокс, коллекция неупорядоченная, а внутри все будет по-порядку. Тут важно понять, что если вы добавите новый элемент в множество, не факт, что он пойдет в конец.  
  
На самом деле, работая с множеством вообще нельзя полагаться на какой-либо порядок элементов, он может быть любым — именно поэтому множество и неупорядоченная коллекция.  
  
Стоит отметить, что множество может быть реализовано множеством различных способов, например можно использовать хеширование, для еще более быстрого поиска элементов, поэтому подробно реализацию я рассматривать не буду. Скажу лишь, что можно схитрить и использовать наши знания по спискам.  
  
Вообще есть еще упорядоченные множества, множества с повторами (мультимножество), и вероятно должно быть упорядоченное мультимножество.  
  
  
*Теория множеств дается проще, если брать множество котят*

#### **Применение**

Множество идеально подойдет для списка любимых котят, потому что их множество.   
  
Но оно действительно подойдет, потому-что такую коллекцию не нужно сортировать (упорядоченность не важна) и мы легко сможем проверить, находится ли какой-нибудь конкретный кот в этом множестве (скажем у меня 100 котят и любимых я кормлю креветками).

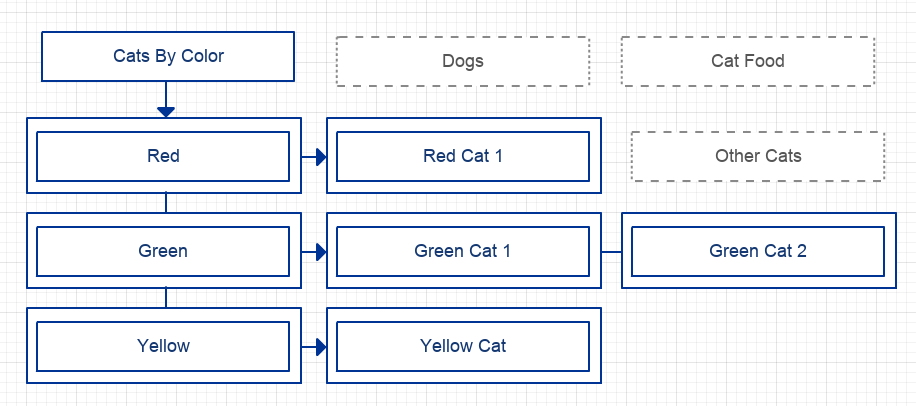
## 4 Словарь (Associative Array, Map, Dictionary)

Словарь (он же ассоциативный массив) — это тот-же вектор, но с небольшими отличиями. В качестве индекса (который в словаре будет называться ключ) могут выступать не только числа, но и любые другие типы данных (даже другие коллекции!). Также допустимы пропуски, если мы все-таки будем использовать в качестве ключа целое число, например у нас может быть элемент связанный с ключем 5, но при этом отсутствовать элемент связанный с ключем 4.  
  
Что все это значит на практике? Всего-лишь, то, что в квадратных скобках для ображения к элементу по “индексу” мы можем указывать произвольный тип, например allMyCats[“Murka”].

#### **Реализация**

Невооруженным видно, что можно просто завести массив (или список) пар (Ключ, Значение) и добавить специальную функцию, которая будет пробегать по этому списку и возвращать определенное значение по связанному с ним ключу.  
  
Мы также не можем сказать какая пара первая, какая последняя и что раньше “Murka” или “Borka”, поэтому словарь считается неупорядоченной структурой.  
  
Опять-же с каждым ключем может быть связано лишь одно значение, поэтому для приведенного примера с именами кошек словарь в чистом виде подходит слабо.  
  
Реализация, как и в случае со множеством, может быть совершенно различной, можно упорядочить пары по ключу и использовать для получения элемента бинарный поиск (в таком случае элементы должны быть упорядочеваемыми). Опять-же можно реализовать словарь с помощью хеширования ключа, что довольно часто используется со строками.

#### **Применение**

Самый правдоподобный и грамотный способ — использовать словарь вместе со списком, где ключем словаря будет строка — имя кошки, а значением — список кошек с таким именем. Это позволит быстро найти всех кошек по имени Мурка и выбрать из них ту, которая в данный момент нужна.  
  
  
*Примерно так выглядит в памяти std::map<std::color, std::list<std::cat>>*

## 5 Стек (Stack)

Есть еще пара структур данных, которые представляют коллекции.  
  
Итак стек — коллекция с необычным доступом, точнее с необычными правилами относительно того, как могут быть добавлены и удалены элементы.  
  
Все просто — добавляемый элемент, называемый “последним”, первый выбывает из из стека.  
  
Стек очень нужен и полезен в программировании. Например с помощью стека осуществляется вложенный вызов процедур — в стек сохраняются адрес возврата и аргументы вызванной функции.

#### **Реализация**

В высокоуровневой реализации ничего особенно интересного нет — указатель на список и элементы добавляются в начало этого списка, и удаляются с него-же.  
  
В низкоуровневой реализации (точнее то, как он реализован в современных архитектурах) есть интересные моменты.  
  
Стек там является небольшим зарезервированным участком памяти и совместно с ним хранится два указателя — на начало стека (где лежит первый доавленный элемент) и конец стека — где лежит последний добавленный.  
  
Если в стек поместить слишком много данных программа завершится со всем знакомой ошибкой — Stack Overflow, это значит, что указатель на конец стека превысил верхний допустимый предел.  
  
Также может случиться обратная ситуация (Stack Underflow), если попытаться забрать из стека больше чем в нем есть, но в высокоуровневых языках она не встречается (понятно почему — нам не дают напрямую работать со стеком).

#### **Применение**

Можно было-бы описать в этом месте стек из бедных котят высотой с гору, но на самом деле в высокоуровневых языках стек редко необходим, часто хватает рекурсии, которая использует стек неявно.

## Разное

На самом деле есть еще куча коллекций, таких как очередь, двусторонняя очередь (дек), двусвязанный список, кольцевое множество, очереди с приоритетом.  
  
Есть деревья и графы.  
  
Есть вероятностные структуры данных, такие как вероятностное множество и список с пропусками.

#### **Строки**

В первую очередь то, как реализованы строки в некоторых языках может показаться странным. Самое простое и эффективное решение это наверное решение C — строка это набор символов, с нулевым символом в конце, что позволяет обходиться без дескриптора.  
  
В C++ std::string уже больше походит на вектор.  
  
Ну а в старом паскале дескриптор (точнее всего-лишь длина) хранится в нулевом элементе массива.  
  
В Haskell String — это список символов ([Char]), из чего вытекает, что получение длины строки имеет сложность O(n). Зато их очень удобно оббегать рекурсивно.  
  
В общем случае, строка — это упорядоченный набор символов и не более. Какой именно тип коллекции будет использован — не важно (ну я бы не советовал использовать множество, ха!).

#### **Очередь (Queue)**

Очередь очень похожа на стек и в тоже время является его противоположностью — первым мы получим обратно не тот элемент, что мы добавили последним, а тот, что “стоит в очереди” дольше всех. Очередь очень удобная структура, но несмотря, на то, что принцип ее работы схож со стеком, в эффективной реализации есть небольшое отличие.  
  
Для стека мы могли схитрить и выделить приемлемый по размеру участок памяти, в случае чего его расширяя, потому-что стек то уменьшается, то увеличивается, т.к. элементы и добавляются и удаляются “с одного конца”. Если же мы представим работу очереди, то она будет “ползти в памяти” — начало будет постоянно сдвигаться вверх, поэтому трюк, который применим для стека, будет работать хуже и тут уже намного лучше будет использовать двусвязный список (и не забудьте хранить указатели на первый и последний элементы).  
  
Еще можете попробовать реализвать очередь на двух стеках, но это тоже менее эффективно.  
  
Также есть дек (двусторонняя очередь — deque). В ней можно добавлять элементы как в конец, так и в начало. И забирать их тоже и с конца и с начала.

Источники:

* <https://unetway.com/tutorial/c-abstrakcia-dannyh>
* <https://habr.com/ru/post/232009/>