Вылиток А.А.

Абстрактный тип данных

Общие сведения

Абстрактный тип данных (АТД) – это тип данных, который предоставляет для работы с элементами этого типа определённый набор функций или операций. Вся внутренняя структура такого типа спрятана от использующего его разработчика программного обеспечения, то есть мы абстрагируемся от его внутреннего устройства при использовании. В C++ АТД реализуется классом, в котором нет открытых членов-данных.

Цель – реализовать и протестировать новый класс, представляющий абстрактный тип данных, т.е. такой, пользователи которого могут обращаться только к публичным операциям, предусмотренными создателем класса, и не имеют информации о внутреннем его устройстве.

Средство – язык C++

Пользователем (клиентом) класса считается не "пользователь программы" или "компьютерный пользователь" вообще, а программист, использующий ваш класс для решения своих задач, пишущий свою программу на его основе. Иногда также под пользователями класса понимаются другие классы, фрагменты программ, которые пользуются его услугами.

Дополнительные требования и условия к Task2

По заданию **требуется** использовать язык C++ (а не C), старайтесь использовать средства языка C++ (ввод-вывод cin/cout и т.п.; new и delete, а не malloc/free).

В данном задании **не разрешается** использовать контейнеры из STL языка C++. Написание собственных аналогов стандартных STL контейнеров (например, string и vector) с соответствующим набором функций приветствуется.

Методы (нестатические), которые не изменяют состояние объекта, должны быть объявлены как константные, если это не противоречит смыслу использования АТД.

Программа должна быть представлена минимум **тремя** файлами:

- <АТД>.h – объявлено все, что необходимо включить в программу, для использования вашего класса АТД,
- <АТД>.cpp – реализации нетривиальных методов класса,
- main.cpp – пользовательский интерфейс для тестирования.

При **тестировании** в поток stderr выводятся результаты теста или ошибки, полученные в ходе теста (для заранее заданного набора тестов также должен выводиться номер теста). Makefile (или аналог) должен содержать цель clean для уборки за собой, а также цель test, которая компилирует main.cpp и прогоняет заранее заданный набор тестов и завершает программу (это можно реализовать с помощью argv). По умолчанию подразумевается цель test.

Для класса, реализующего АТД, необходимо **определить**:

- конструктор по умолчанию,
- конструктор копирования (глубокое копирование, если необходимо),
- операцию присваивания (предварительно уничтожить старое значение + см. требование выше относительно копирования),
- деструктор (стоит предусмотреть ситуацию повторного вызова деструктора: он не должен пытаться в этом случае освобождать в динамической памяти объекты повторно – это может привести к ошибке выполнения),
- базовый класс исключения, выбрасываемого всеми операциями класса, возможно с потомками – для более детальной классификации ошибок, – вложен в основной класс (например, string_list::error).

Правильный вывод ошибок должен содержать, хотя бы приблизительно, номер строки, в которой произошла ошибка.

Варианты задания (сложность указана звездочками, некоторые варианты предложены П.Г. Сутыриным)

Под строкой понимается набор символов, завершающийся символом ’\0’, длина строки ограничена доступной оперативной памятью.

Вариант 1. Список строк (String list)

Вариант 1.1 (*)

Строки **сохраняют** начальные и конечные пробельные последовательности.

Обязательные возможности класса:

1) list(char*) – конструкторы списка из одного элемента;
2) list + list – вернуть конкатенацию списков, операнды неизменны;
3) list += list – присоединить к первому списку, второй неизменен;
4) char* + list – добавить элемент в начало списка;
5) list + char* – добавить элемент в конец списка;
6) list - char* – удалить элемент, если есть;
7) list[i] – получить копию i-го элемента;
8) приведение к char* – вернуть строку, где все элементы разделены символом ’\n’;
9) int length() – текущий размер.

Вариант 1.2 (**)

Расширение варианта 1.1.
Реализовать независимость всех операций от пробелов в начале или в конце строк, т.е. можно добавить "abc ", затем успешно удалить " abc " (при этом при хранении строки по-прежнему **сохраняют** начальные и конечные пробельные последовательности).

Дополнительные функции:

10) list * list – для списков одной длины построить список пар через \n, по одной строке каждого списка;
11) contains(char*, bool space=false) – проверка вхождения строки в список, space – учитывать ли пробелы;
12) перегрузка операции << – вывод элементов списка через ’\n’.

Вариант 1.3 (***)

Расширение варианта 1.2.
Дополнительные функции:

13) list.insert(char*, i) – вставить элемент на i-е место, со сдвигом остальных вперед;
14) list[i] = char*– новое значение для i-го элемента;
15) delete list[i] – перегрузка операции delete (удалить i-й элемент);
16) перегрузка операции >> – добавление нового элемента в список.

Вариант 2. Стек строк

Стек в обычном понимании (LIFO, последним вошел – первым вышел).

Вариант 2.1 (*)

Строки **сохраняют** начальные и конечные пробельные последовательности.

Обязательные операции:

1) stack(char*) – конструкторы стека из одного элемента;
2) stack::push(char*) и синоним stack + char*– положить копию строки;
3) char* stack::pop() – вернуть верхний элемент с удалением из стека;
4) char* stack::peek() – вернуть копию верхнего элемента без удаления;
5) int stack::length() – текущий размер;
6) stack - char* – синоним pop, записывающий результат во второй операнд;
7) stack1 + stack2 – вернуть конкатенацию стеков, при этом на вершине стека должен оказаться, последний элемент второго стека, операнды неизменны;
8) stack1 += stack1 – присоединить к первому списку, второй неизменен;
9) приведение к char*– вернуть строку, где все элементы разделены \n;
10) int stack::length() – текущий размер.

Вариант 2.2 (**)

Расширение варианта 2.1.
Реализовать независимость всех операций от пробелов в начале или в конце строк, т.е. можно добавить "abc ", затем успешно удалить " abc " (при этом при хранении строки по-прежнему сохраняют начальные и конечные пробельные последовательности).

Дополнительные функции:

11) contains(char*, bool space= false) – проверка вхождения строки в список, space – учитывать пробелы или нет, вернуть позицию в стеке, если элемента нет вернуть -1;
12) перегрузка операции << – вывод элементов списка через \n;
13) перегрузка операции >> – добавление нового элемента в стек.

Вариант 3. Множество строк

Эффективный для поиска тип данных «множество строк». Чаще всего в этом множестве будет осуществляться поиск, поэтому реализовать его можно бинарным деревом поиска. Пользователя нужно избавить от деталей реализации, предоставив простой внешний интерфейс.

Вариант 3.1 (**)

Строки **не сохраняют** начальные и конечные пробельные последовательности. Исходно вставка всегда осуществляется быстрейшим способом, без балансировки. Реализовать независимость всех операций от пробелов в начале или в конце строк, т.е. можно добавить " abc", затем успешно удалить "abc ".

Обязательные операции:

1) set(char*) – конструкторы множества из одного элемента;
2) set::add(char*) и синоним set + char*– вставка с балансировкой;
3) set:: contains (char*) и синоним set ^ char*– проверка вхождения строки в множество;

4) set::delete(char*) и синоним set - char*– удалить, если есть;
5) int set::size() – число элементов;
6) set::optimize() – балансировка дерева;
7) set1+set2 – объединение множеств, операнды неизменны;
8) set1^set2 – пересечение множеств, операнды неизменны.

Вариант 3.2 (***)

Расширение варианта 3.1
Дополнительные функции:

9) set1/set2 – множество элементов set1, не включающее элементы из set2, операнды неизменны;
10) перегрузка операции << – вывод элементов списка через '\n';
11) перегрузка операции >> – добавление нового элемента в множество;
12) set1 += set2 – присваивание с объединением, правый операнд неизменен;
13) set1 ^= set2 – присваивание с пересечением, правый операнд неизменен;
14) set1 /= set2 – присваивание с исключением, правый операнд неизменен;
15) set1*set2 и set1 *= set2 – симметричная разность множеств.

Вариант 4. Вещественная матрица

Вещественным числом считается число типа double. Необходимо реализовать вещественную матрицу – понятие линейной алгебры. Нужно обеспечить пользователя класса естественным интерфейсом математических операций для работы с матрицами и составления вычислительных программ.
Для удобства тестирования и отладки ввести строковое представление матрицы, похожее на инициализатор двумерного массива. Так, строка "{{1, 0, 0}, {0, 1, 0.5}}" обозначает матрицу размера 2×3:

```
| 1   0   0 |
| 0   1 0.5 |
```

Вариант 4.1 (***)

Обязательные операции:

1) matrix(int n, int m) – конструктор матрицы размера n×m со значениями 0.0;
2) matrix(double) – матрица 1×1 с этим элементом;
3) matrix(double*, int m) – матрица-строка из массива длины m;
4) matrix(int n, double*) – матрица-столбец из массива длины n;
5) matrix(char*) – из строкового представления (см. выше);
6) static matrix matrix::identity(int n) – возвращает единичную матрицу размера n;
7) static matrix matrix::diagonal(double* vals, int n) – возвращает диагональную матрицу размера n с заданными элементами по главной диагонали;
8) int matrix::rows() – число строк;
9) int matrix::columns() – число столбцов;
10) matrix::set(int i, int j, double val) – присвоить значение элементу [i][j];
11) matrix matrix::matrix[i] – i-я строка в виде новой матрицы, если такая строка есть – 1-й приоритет
12) matrix matrix::matrix[j] – j-й столбец в виде новой матрицы, если такой столбец есть – 2-й приоритет, – иначе ошибка (если M – матрица, то M[i] – матрица из одной строки, а M[i][j] – матрица 1×1 из одного элемента);
13) matrix * scalar и matrix*=scalar – умножение матрицы на скаляр;
14) перегрузка операции << – вывод матрицы, в привычном двумерном виде.

Вариант 4.2 (****)

Расширение варианта 4.1.
Все арифметические операции выбрасывают исключение, если матрицы несовместимы по размерам.
Дополнительные функции:

15) matrix + matrix ;
16) matrix += matrix ;
17) matrix - matrix ;
18) matrix -= matrix ;
19) matrix * matrix ;
20) matrix *= matrix ;
21) -matrix – унарный минус, применить ко всем элементам ;
22) matrix == matrix – точность сравнения задана статической константой matrix::EPS ;
23) matrix != matrix ;
24) matrix | matrix – конкатенировать (приписать) матрицы вертикально (вторую справа от первой);
25) matrix / matrix – конкатенировать (приписать) матрицы горизонтально (вторую под первой).

Вариант 4.3 (*****)

Расширение варианта 4.2.
Дополнительные функции:

26) ~matrix – обратная матрица;
27) double matrix::determinant() – определитель;
28) matrix matrix::solve() – решить любым способом неоднородную систему из n линейных алгебраических уравнений с n неизвестными, представленную матрицей размера n×(n+1); вернуть решение в виде матрицы 1×n; выбросить исключение в случае неразрешимости системы (с учетом EPS); обязательно зафиксируйте ошибку, например: x = (A | b).solve(); if(A*x != b) error("solve");
29) эффективная реализация индексирования – обеспечение возможности присваивания m[i][j] = вещественное_значение; (для реализации понадобится внутренний класс-адаптер «Строка_матрицы» с внутренним полем-ссылкой на «настоящую» строку матрицы).

Вариант 5. Функции

Обязательно придумайте лаконичное строковое представление (наподобие того, что для матриц), сделайте конструктор из него и приведение к нему.

Вариант 5.1 (***) Кусочно-постоянная функция (piecewise constant function)

Вещественная функция (double), определенная на всей числовой прямой. Область определения делится на N промежутков (первый и последний – полуинтервалы), внутри каждого из которых функция принимает некоторое постоянное для этого промежутка значение. На границах всех промежутков непрерывна либо слева, либо справа (выбрать).

Обязательные операции:

1) function(char*) – определение функции из строкового представления;
2) function1 + function2 ;
3) function1 - function2 ;
4) function1 * function2 ;
5) function1 / function2 ;
6) function1 += function2 ;

7) function1 -= function2 ;
8) function1 *= function2 ;
9) function1 /= function2 ;
10) function(double x) – значение функции в точке – операция ();
11) integral(double x) – численное интегрирование на отрезке;
12) перегрузка операции << – вывод функции в удобном для чтения виде.

Вариант 5.2 (****) Многочлен (polynomial function)

Понятие алгебры. Многочлен степени n от одной вещественной (double) переменной x. Конструктор, принимающий на вход строковое представление, должен понимать любой порядок распределения степеней (строка вида "x^2 – x + 1" эквивалентна строкам "-x + x^2+1" и "-x+1+x^2"). Преобразование к строке выдает многочлен, упорядоченный по степеням x, конструктор воспринимает любой их порядок. Простейшим примером многочлена является числовая константа (многочлен степени 0).

Обязательные операции:

1) function(char*) – определение функции из строкового представления;
2) function1 + function2 ;
3) function1 - function2 ;
4) function1 * function2 ;
5) function1 / function2 – деление многочленов с отбрасыванием остатка;
6) function1 % function2 – остаток от деления многочленов;
7) function1 += function2 ;
8) function1 -= function2 ;
9) function1 *= function2 ;
10) function1 /= function2 – деление многочленов с отбрасыванием остатка
11) function1 %= function2 – остаток от деления многочленов;
12) function(double x) – значение функции в точке, реализовать через операцию () ;
13) символьное интегрирование (с построением нового многочлена) ;
14) перегрузка операции << – вывод многочлена, в удобном для чтения виде.

Вариант 5.3 (*****) – Многочлен (polynomial function)

Расширение варианта 5.3.
Реализовать упрощение многочлена (приведение подобных членов) либо при преобразовании к строке, либо поддерживать всегда в приведенном виде.

Дополнительные функции:

15) символьное дифференцирование (с построением нового многочлена);
16) определенный интеграл в заданных пределах (возвращает число);
17) перегрузка операции >> – ввод многочлена с присваиванием.

Варианты 6. Граф

Релизовать текстовое представление графа (в виде строки). Например: "((1, 2, 3, 4), ((1,'a', 2),(2,'abc', 3)))" задает граф с вершинами 1, 2, 3 и двумя ребрами: с пометкой "a" из 1 в 2 и с пометкой "abc" из 2 в 3, вершина 4 изолирована. Придумайте вспомогательные классы "вершина", "ребро (дуга)", которые бы незаметно для пользователя конструировались из строк, превращались в строки, и т.д.

Вариант 6.1 (****) Ориентированный граф

Обязательные операции:

1) добавление (add – одна функция с разным набором аргументов): вершины, ребра, дуги;

2) удаление (remove – одна функция с разным набором аргументов): вершины, ребра, дуги;
3) вывод списка дуг, исходящих из вершины;
4) операция + (объединение графов, добавление вершины, добавление ребра на те же вершины или вместе с одной новой вершиной или с двумя новыми вершинами, если их еще нет в графе);
5) операция – (минус) – удаление из графа (вершины, ребра, дуги);
6) операция delete[] – в качестве синонима для удаления вершины;
7) проверка достижимости одной вершины из другой;
8) перегрузка операции << – вывод графа, в удобном для чтения виде;
9) поиск сильно связных компонент.

Вариант 6.2 (*****) Ориентированный граф с петлями

Расширение варианта 14.
Дополнительные функции:

10) поиск всех возможных путей из вершины в вершину (по каждому ребру можно проходить ровно 1 раз) – придумать класс + текстовое представление пути;
11) поиск циклов в графе;
12) построение остовного дерева (любого).

Вариант 6.3 (******) Неориентированный граф с петлями, мультидугами и весами на дугах

Расширение варианта 6.2.
Дополнительные функции:

13) поиск оптимальных путей из вершины в вершину – минимальных по стоимости – придумать класс + текстовое представление пути;
14) построение остовного дерева – минимального по стоимости;
15) вершинная/реберная раскраска;
16) поиск циклов в графе.

Вариант 7. Дата и время (datetime)

Реализовать классы для работы с моментами времени и интервалами времени**.** Текстовое представление – например, ISO формат.

Вариант 7.1 (**)

Допускается ориентирование на 30 дней в каждом месяце, и игнорирование високосных лет.

Обязательные операции:

1) конструкторы и преобразования для (более чем одного) вариантов текстового момента времени, конструкторы и преобразования минимум для двух вариантов представления интервала:
   - интервал: "2-1-15 00:05:00.000" – два года, полтора месяца
   - интервал “2018-02-18T17:16:32.829000 - 2019-02-19T17:16:32.829000” – год и 1 день ;
2) получить текущий момент времени;
3) moment - moment = interval ;
4) moment + interval = moment ;
5) moment - interval = moment ;
6) interval + interval = interval ;
7) interval - interval = interval ;
8) интервал * double ;
9) перегрузка операции << – вывод, в удобном для чтения виде (желателен ISO формат).

Вариант 7.2 (***)

Расширение варианта 7.1.
Полностью корректное определение моментов времени, с учетом високосных лет и количества дней в месяцах.

Дополнительные функции:

10) интервал / double ;
11) конструктор/преобразование для типа unix timestamp ([milli]seconds since Epoch), вариант seconds или milliseconds обязательным аргументом на выбор пользователю ;
12) interval += interval;
13) interval -= interval;
14) moment += interval,;
15) moment -= interval;
16) перегрузка операции >> – ввод момента или интервала с присваиванием.