# Система управления пользователями

Реализуйте классы User и Group. Класс User должен содержать информацию, такую как имя пользователя, уникальный идентификатор и другие релевантные данные (на ваше усмотрение), а также содержать ссылку на группу, в которой состоит пользователь (пользователь может и не состоять в группе). Класс Group должен содержать идентификатор группы и список всех пользователей, которые в ней состоят. **Между классами User и Group не должно быть циклических зависимостей!**

Создайте консольную утилиту для управления пользователями и группами пользователей, которая должна поддерживать следующие команды:

* createUser {userId} {username} {…дополнительная информация…} – создание нового пользователя;
* deleteUser {userId} – удаление пользователя;
* allUsers – вывод информации по всем пользователям;
* getUser {userId} – вывести информацию по одному пользователю;
* createGroup {groupId} – создать новую группу;
* deleteGroup {groupId} – удалить группу;
* allGroups – вывеси информацию по всем группам, включая всех пользователей, которые в них состоят;
* getGroup {groupId} – вывести информацию по одной группе, включая всех пользователей, которые в ней состоят.

# Реализовать type list

Реализуйте класс или структуры с именем TypeList, представляющую собой упорядоченную коллекцию типов. Реализуйте следующие методы работы с TypeList (в виде шаблонизированных структур или constexpr функций):

* Получение элемента списка по его индексу (попытка обращения к элементу, которого не существует, должна приводить к ошибке компиляции);
* Получение размера списка;
* Проверка наличия типа в списке (constexpr bool);
* Получение индекса типа в списке;
* Добавление типа в конец списка;
* Добавление типа в начало списка.

Все детали реализации следует скрыть в отдельном пространстве имен. При написании следует использовать variadic templates.

Для проверки работоспособности реализованных методов напишите тестовый код, в которым результаты применения методов будут проверятся при помощи static\_assert и std::is\_same.

# Реализовать type map

Разработайте шаблонный контейнер TypeMap с использованием ранее реализованного TypeList. TypeMap должен представлять собой ассоциативный контейнер, где ключами являются типы, а значениями - соответствующие объекты.

Класс TypeMap должен обеспечивать следующие операции:

* Добавление элемента в контейнер с указанием типа в качестве ключа.
* Получение значения по заданному типу ключа.
* Проверка наличия элемента по типу ключа.
* Удаление элемента по типу ключа.

Ниже приведен пример использования:

|  |
| --- |
| struct DataA {  std::string value;  };  struct DataB {  int value;  };  int main() {  TypeMap<int, DataA, double, DataB> myTypeMap;  // Добавление элементов в контейнер  myTypeMap.AddValue<int>(42);  myTypeMap.AddValue<double>(3.14);  myTypeMap.AddValue<DataA>({"Hello, TypeMap!"});  myTypeMap.AddValue<DataB>({10});  // Получение и вывод значений по типам ключей  std::cout << "Value for int: " << myTypeMap.GetValue<int>() << std::endl; // Вывод: 42  std::cout << "Value for double: " << myTypeMap.GetValue<double>() << std::endl; // Вывод: 3.14  std::cout << "Value for DataA: " << myTypeMap.GetValue<DataA>().value << std::endl; // Вывод: Hello, TypeMap!  std::cout << "Value for DataB: " << myTypeMap.GetValue<DataB>().value << std::endl; // Вывод: 10  // Проверка наличия элемента  std::cout << "Contains int? " << (myTypeMap.Contains<int>() ? "Yes" : "No") << std::endl; // Вывод: Yes  // Удаление элемента  myTypeMap.RemoveValue<double>();  // Попытка получения удаленного элемента  std::cout << "Value for double after removal: " << myTypeMap.GetValue<double>() << std::endl; // Вывод: (некорректное значение)  return 0;  } |

# Операторные MixIn

1. Реализуете MixIn класс less\_then\_comparable, который при помощи CRTP «подмешивает» в целевой класс операторы сравнения (>, <=, >=, ==, !=).
2. Реализуйте MixIn класс counter, который обеспечивает возможность подсчета созданных экземпляров целевого класса.

Далее приведен пример использования созданных MixIn:

|  |
| --- |
| class Number: public less\_than\_comparable<Number>, public counter<Number> {  public:  Number(int value): m\_value{value} {}  int value() const { return m\_value; }  bool operator<(Number const& other) const {  return m\_value < other.m\_value;  }  private:  int m\_value;  };  int main()  {  Number one{1};  Number two{2};  Number three{3};  Number four{4};  assert(one >= one);  assert(three <= four);  assert(two == two);  assert(three > two);  assert(one < two);  std::cout << "Count: " << counter<Number>::count() << std::endl;  return 0;  } |

# Паттерн Singleton

Используя паттерн Singleton, разработайте систему протоколирования событий в системе. Система должна: - поддерживать 3 уровня важности событий (нормальный, замечание, ошибка); - обеспечить фиксацию события (с событием фиксируются время, важность, текстовое сообщение); - выводить на печать 10 последних событий.

Пример использования:

|  |
| --- |
| #include “log.h”  void main(void) {  Log \*log = Log::Instance();  log->message(LOG\_NORMAL, “program loaded”);  …  log->message(LOG\_ERROR, “error happens! help me!”); log->print();  } |

# Порождающие паттерны

Трасса трофи-рейда представляется в виде последовательности контрольных пунктов КП. Бывают два вида КП: обязательного и необязательного взятия. Про каждое КП хранится следующая информация:

* имя кп (строка);
* координаты: широта (число с плавающей точкой в диапазоне -90.0°…+90°) и долгот а (число с плавающей точкой в диапазоне -180°…+180°).

Для КП с необязательным взятием также хранится значение штрафа за пропуск этого кп (число с плавающей точкой, представляющее время в часах). Используя подход, применяемый в паттерне Builder, разработайте фрагмент системы, обеспечивающий обработку списка КП. Реализуйте ConcreteBuilder для:

1. Вывода списка КП в текстовом виде. Для каждого КП должны выводится: порядковый номер; имя; координаты; время штрафа или строка «незачёт СУ» для обязательных КП.
2. Подсчёта суммарного штрафа по всем необязательным КП.
3. Вывода списка КП в виджете с таблицей (например, QTableView или аналогичном).

# Паттерн Bridge

Используя паттерн Bridge реализуйте объект «множество», которое представляется различными структурами данных в зависимости от числа элементов. Ваша реализация должна включать:

* «абстракцию» - класс Множество, имеющий основные операции для работы с множеством (добавить элемент, удалить элемент, проверить наличие элемента, объединение и пересечение множеств);
* «абстрактную реализацию» - интерфейс объектов, обеспечивающих хранение множеств. В нём может быть объявлены функции для доступа к данным, отличающиеся от объявленных в классе Множество;
* как минимум две конкретных реализации, обеспечивающих хранение множеств в разных структурах данных. Например, простым массивом для небольшого числа элементов и деревом/хэш-таблицей для большого.
* Множество должно менять используемую реализацию в зависимости от числа хранимых элементов.

# Компоновщик и Приспособленец

Используя подход, предлагаемый паттернами компоновщик и приспособленец, реализуйте фрагмент системы классов для представления арифметических выражений.

Система должна включать классы для нескольких арифметических операторов, переменных (хранит имя переменной) и констант (хранит значение). Эти классы должны обеспечить представление выражения в виде дерева из операторов с переменными или константами в листьях.

В классах операторов, констант и переменных должны быть реализованы функции для:

* печати выражения;
* вычисления значения выражения. Этой функции в качестве параметра передаётся std::map в которой хранятся значения для всех переменных из выражения.

Переменные и константы должны быть реализованы в виде приспособленцев – если в выражении несколько раз встречается одинаковая переменная или константа, все её вхождения должны быть реализованы в виде одного объекта. Следует разработать фабрику, которая должна иметь методы для создания и удаления объектов переменных и констант. В фабрике должны быть заранее созданы (и никогда не удаляться) объекты для констант от -5 до 256 (так сделано в языке Python). Остальные объекты создаются по требованию и удаляются когда перестают использоваться. Для удаления объектов в фабрике должны быть предусмотрены соответствующие функции.

Ниже представлен пример, показывающий использование такой системы классов для вычисления значения выражения 2 + 𝑥 при 𝑥 = 3.

|  |
| --- |
| ExperssionFactory factory;  Constant \*c = factory.createConstant(2);  Variable \*v = factory.createVariable("x");  Addition \*expression = new Addition(c, v);  map context; context["x"] = 3;  cout << expression->calculate(context) << endl;  delete expression; // Все "нижележащие" объекты должны  // быть освобождены деструктором. |