Домашнее задание по теме ”Шаблоны“

Умный указатель

Умный указатель - это специальная оболочка над указателем, которая самостоятельно освобождает память. Напишите класс умного указателя `SmartPointer`. Нужно написать:

- Конструктор
- Деструктор, который будет освобождать память
- Перегруженный унарный оператор `*`, который будет возвращать значение (по ссылке)

Пример работы с таким указателем:

```
#include <iostream>
#include <string>
using namespace std;

// Тут вам нужно написать шаблонный класс SmartPointer

int main()
{
    SmartPointer<int> pi = new int(123);
    SmartPointer<string> ps = new string("Hello");

    *pi = 543;
    *ps = "World";

    cout << *pi << "  " << *ps << endl;

    // Освобождать не нужно, так как всё освободится в деструкторе
    // Таким образом, память всегда освободится
}
```

Пара

Пара - это шаблонный класс, который содержит 2 объекта. Нужно написать:

- Конструктор

Пример работы с таким шаблонным классом:

```
#include <iostream>
#include <string>
using namespace std;

// Тут вам нужно написать шаблонный класс Pair
// ...

int main()
{
    Pair<int, string> a = {777, "Axolotl"};
    Pair<string, float> b = {"Hippo", 6.45};

    cout << a.first << "  " << a.second << endl;
    b.first += "potamus";
    b.second = 3.14;
    cout << b.first << "  " << b.second << endl;
}
```

Динамический массив (аналог std::vector)

Динамический массив - массив, который сам расширяется при добавлении в него элементов. Он по умолчанию реализован в разных языках программирования. В частности, в языке C++ это шаблонный класс `std::vector` (`vector` - не совсем удачное название, так как можно подумать, что этот массив имеет какое-то отношение к математическим векторам, но это не так). Для работы с ним нужно подключить библиотеку `<vector>`. Но в этом задании мы рассмотрим поэтапное создание своего динамического массива. Исходный код – в папке `handmade_dynarray`.

Часть 0: Динамический массив на языке C

В файле `0handmade_dynarray.c` содержится исходный код для динамического массива на языке `C`. Такой мы писали в прошлом семестре, когда реализовывали стек на основе динамического массива. Также там написаны функции для работы с этим динамическим массивом. Функция `dynarray_push_back` – добавляет элемент в конец массива. Обратите внимание, что для хранения размеров и индексов массива используется специальный целочисленный тип `size_t`. Это специальный тип, который задаётся в стандартных библиотеках `C` и `C++` для хранения индексов. Обычно это просто синоним типа `unsigned int` или `unsigned long`.
Теперь будем поэтапно переписывать эту структуру данных с языка `C` на язык `C++`.

Часть 1: Инкапсуляция

Сначала нужно все функции для работы с динамическим массивом сделать методами класса `Dynarray`. К примеру функция:

```
void dynarray_push_back(Dynarray* pd, Data x)
```

переходит в метод класса:

```
void push_back(Data x)
```

Теперь работать с массивом стало намного удобней.

Часть 2: new / delete

В языке `C++` следует всегда предпочить операторы `new/delete` функциям `malloc/free`. Поэтому в этой часть мы поменяем все `malloc`-и на `new`, а `free` изменим на `delete`. Аналога `realloc` нет, поэтому просто сами перевыделяем память. Чтобы проверить `malloc` на правильность работы нужно сравнить его возращаемое значение с нулём. Проверка `new` на правильность работы выполняется с помощью исключений. Пока мы эту тему не прошли, так что просто ничего не проверяем.
Также в этой части мы меняем все вызовы `printf` на `std::cout <<`.
Для того чтобы скопировать элементы из одного массива в другой используем стандартную функцию `std::copy_n`.

Часть 3: Конструкторы и деструктор.

Вызов функций `init` и `destroy` при каждом создании/удалении объекта кажется не очень хорошей идеей. Если программист забудет вызвать их, то в программе возникнет ошибка или утечка памяти. Эти функции должны быть частью процесса создания/удаления объекта и должны вызываться автоматически. Перепишем эти функции в конструктор `Dynarray` и деструктор `~Dynarray` соответственно.

Часть 4: Шаблоны.

В качестве хранимого типа мы используем `Data`, который задаём с помощью `typedef`:

```
typedef int Data;
```

Таким образом, можно изменять тип данных в массиве, но нельзя, например, создать 2 динамических массива с разными типами данных в одной программе. Используем шаблоны, чтобы добиться нужного результата.

Часть 5: private / public.

Программист, который будет пользоваться текущей реализации нашего массива, может легко его сломать. Например так:

```
Dynarray<int> a;
a.size = 100000;
```

Чтобы минимизировать количество ошибок, которые могут возникнуть при работе с нашим классом, скроем поля, изменение которых может всё поломать (то есть все поля). Поля `size`, `capacity` и `values` помещаем в `private`. Так как мы всё-таки хотим дать программисту возможность знать эти значения, введём публичные методы `get_size()` и `get_capacity()`. Для работы с элементами массива введём функцию `at`, которая будет работать как `operator[]`, но проверять входной индекс на правильность.

Часть 6: initializer_list конструктор.

В текущей реализации нельзя инициализировать значения нашего динамического массива также как мы делали с обычным массивом. Вот так:

```
Dynarray<int> a = {4, 8, 15, 16, 23, 42};
```

Чтобы добавить такую возможность в наш класс нужно добавить конструктор, который будет принимать специальный объект типа `std::initializer_list<T>`. Для копирования элементов из этого объекта в наш массив используем стандартную функцию `std::copy`.

Часть 7: Оператор присваивания (operator=).

Если не перегрузить оператор присваивания, то компилятор автоматически создаст свой (который будет просто копировать значения всех полей). В нашем случае это очень плохо, потому что при присваивании будет просто копироваться значение указателя `values`, а не сами элементы выделенные в динамической памяти.
Одна тонкость, которую нужно учесть при перегрузке этого оператора – это случай `a = a`, то есть когда элемент присваивается самому себе.

Часть 8: Итераторы. range-based цикл for.

В языке `C++` есть удобная версия цикла `for`, которая выглядит так:

```
Dynarray<string> a = {"Cat", "Dog", "Nutria", "Echidna", "Turtle", "Coati"};
for (string s : a)
        cout << s << endl;
```

Но чтобы можно было работать с таким циклом нужно, чтобы наш класс содержал методы `begin()` и `end()`, которые бы возвращали итератор. Итератор – это объект особого типа с операцией унарная звёздочка (`operator*`) и с возможностью прибавлять/удалять целые числа. В нашем случае это просто указатель. Однако ничто не мешает создать свой класс для итератора и перегрузить соответствующие операторы.

Связный список

В папке `handmade_list` лежит реализация односвязного списка на языке `C`. Ваша задача – переписать эту структуру на язык `C++`, пройдя те же шаги, что были пройдены для динамического массива выше. Тонкий момент – согласно принципу инкапсуляции структуру Node нужно поместить внутрь класса List.

* Словарь - хеш-таблица (Unordered Map)

Написать свой шаблонный класс `UnorderedMap` для словаря, основанного на хеш-таблице. Используйте шаблонный класс `Pair`, чтобы хранить элементы. Можно сделать эту задачу вместо задачи на связный список.

Структуры данных и шаблонные классы, которые их реализуют на языке C++.

Все они содержатся в пространстве имён `std`.

vector	динамический массив
forward_list	односвязный список
list	двусвязный список
priority_queue	очередь с приоритетом
set	бинарное дерево поиска с балансировкой (только ключ)
map	бинарное дерево поиска с балансировкой (ключ-значение)
unordered_set	хеш-таблица (только ключ)
unordered_map	хеш-таблица (ключ-значение)