Семинар #3: Шаблоны (и строки).

Часть 1: Строки в C++

На прошлом занятии мы писали свой класс для строк. Но в языке `C++` есть библиотека `string` (не путать с библиотекой `string.h` из языка `C`), которая предоставляет класс строк `std::string`. В отличии от строк языка C, которые являются просто массивами элеметов типа `char`, строки в языке C++ являются классом. Работать с ними гораздо проще и удобней, чем со строкам в языке `C`.

Строки в `C`

```c
#include <stdio.h>
#include <string.h>

int main () {
    char a[10] = "Deus";
    char b[10];
    strcpy(b, "machina");

    char c[20];
    strcpy(c, a);
    strcat(c, " ex ");
    strcat(c, b);
    printf("%s\n", c);

    if (strcmp(b, a) > 0) {
        printf("b is greater\n");
    }
}
```

Строки в `C++`

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

int main () {
    string a {"Deus"};
    string b;
    b = "machina";

    string c = a + " ex " + b;
    cout << c << endl;

    if (b > a) {
        cout << "b is greater" << endl;
    }
}
```

Задачи

- Создайте строку `Hello world` и напечатайте её на экран.
- Создайте программу, которая будет считывать слова (используйте `cin`) в бесконечном цикле и каждый раз печатать сумму всех слов. Например, если пользователь ввёл `Hello`, то программа должна напечатать `Hello` и запросить следующее слово. Если затем пользователь введёт `World`, то программа должна будет напечатать `HelloWorld` и запросить следуещее слово.

Часть 2: Шаблонные функции

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

template <class T>
T getMax (T a, T b) {
    if (a > b)
        return a;
    else
        return b;
}
int main () {
    int a = 5, b = 6;
    cout << getMax<int>(a, b) << endl;

    long long n = 9645634567, m = 7356735634;
    cout << getMax(n, m) << endl;

    cout << getMax(4.6, 5.3) << endl;

    std::string s1 = "deus", s2 = "machina";
    cout << getMax(s1, s2) << endl;
}
```

Задачи

- Написать шаблонную функцию `T triple(T x)`, которая увеличивает переменную в 3 раза. Проверить её на переменных типа `int`, `float`, `Complex`, `std::string`. Реализацию класса `Complex` можно найти в файле `complex.h`.

```cpp
cout << triple<int>(5) << endl;

std::string s = "Hello"
cout << triple(s) << endl;  // HelloHelloHello
```

- Написать шаблонную функцию `T sum(T arr[], int size)`, которая возвращает сумму массива переменных. Проверить её на переменных типа `int`, `float`, `Complex`, `std::string`.

```cpp
int numbers[] = {4, 8, 15, 16, 23, 42};
cout << sum<int>(numbers, 6) << endl;

std::string words[] = {"Deus", "Ex", "Machina"};
cout << sum(words, 3) << endl;   // DeusExMachina
```

Часть 3: Шаблонные классы

Можно создавать не только шаблонные функции, но и шаблонные классы. Например, создадим шаблонный класс статического массива:

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

template <typename T, int size>
class Array {
private:
    T data[size];

public:
    T& operator[](int id) {
        return data[id];
    }
};

int main() {
    Array<int, 10> numbers;
    for (int i = 0; i < 10; ++i) {
        numbers[i] = rand() % 100;
    }
    cout << numbers[1] << endl;
}
```

В файле `8array.cpp` содержится код для такого массива.

Задачи

- Измените оператор взятия по индексу так, чтобы при выходе за пределы массива программа выдавала ошибку и завершалась.
- Добавьте метод `reverse`, который будет обращать статический массив `Array`. Протестировать этот метод на `Array` разного типа.
- Добавьте метод `sort`, который будет сортировать статический массив `Array` (для простоты используйте простую $O(n^2)$ сортировку).
- Перегрузите операторы больше и меньше для этого массива (сравнение в лексиграфическом порядке). Теперь можно будет отсортировать массив массивов.

Часть 4: Основы std::vector

`std::vector` - это удобный динамический массив C++ (`vector` - не совсем удачное название, так как можно подумать, что этот массив имеет какое-то отношение к математическим векторам, но это не так). Он хранит все элементы в куче и автоматически увеличивается в размерах, если нужно. Это шаблонный класс и может хранить в себе почти что угодно. Для работы с ним в стандартной библиотеки языка C++ уже написано множество методов и функций.

Для добавления новых элементов в вектор используйте метод `push_back`. Этот метод добавит элемент в конец и увеличит размер вектора. Метод `size` возвращает размер вектора, а метод `capacity` – его вместимость. Функции вектор передаётся также, как и другие объекты.

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

int main () {
    std::vector<int> v = {54, 62, 12, 97, 41, 6, 73};
    cout << v[1] << endl;

    v.push_back(44);
    cout << "Size = " << v.size() << " Capacity = " << v.capacity() << endl;

    for (int i = 0; i < v.size(); i++) {
        cout << v[i] << ' ';
    }
    cout << endl;
}
```

- **Размер и вместимость:** Проверьте как работает автоматическое расширение вектора. Для этого создайте пустой вектор и заполните его числами от 0 до 300 (используйте `push_back`). При этом на каждом шаге печатайте размер вектора и его вместимость.
- **Reserve:** Постоянные расширения вектора могут быть очень трудозатратны. Используйте метод `reserve`, чтобы расширить вектор до значения 300 перед добавлением элементов. Проверьте как будет меняться размер и вместимость вектора в этом случае.

  ```
  v.reserve(300);
  ```

- **Вектор строк:** Создадим следующий вектор строк:

  ```
  std::vector<std::string> animals = {"Cat", "Dog", "Bison", "Rabbit", "Spider", "Wolf",
      "Turkey", "Lion", "Pig", "Snake", "Shark", "Bird", "Fish"};
  ```

  – Напечатайте этот вектор на экран.
  – Напечатайте только тех животных, которые начинаются на букву `S`.
  – Напишите функцию, `print_by_letter` которая принимает на вход вектор строк и один символ. Она должна печатать все строки, начинающиеся на этот символ.
  – Написать функцию `get_by_letter`, которая принимает на вход вектор строк и один символ. Эта функция должна должна возвращать вектор строки слов, которые начинаются на соответствующий символ. Для этого внутри функции вы должны создать новый вектор, заполнить его нужными строками и вернуть. Проверить правильность работы функции, вызвав её в функции main и напечатав результат.
  – Написать функцию `change_by_letter`, которая принимает на вход вектор строк и один символ. Функция должна заменять все слова, начинающиеся на этот символ на слово ‘‘`Animal`’’.

- **Шаблоны + векторы:** Написать шаблонную функцию `T sum(const std::vector<T>& vec)`, которая возвращает сумму вектора переменных. Проверить её на переменных типа `int`, `float`, `Complex`, `std::string`.

Часть 5: Создаём свой динамический массив

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`. 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`.

Шаг 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++.

Шаг 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)
```

Шаг 2: new / delete

В языке C++ следует всегда предпочить операторы `new/delete` функциям `malloc/free`. Поэтому на этом шаге 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 возвращаемое 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 <<`.

Шаг 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.

Шаг 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.

Шаг 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.

Шаг 6: 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.

Шаг 7: `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`.

Шаг 8: Итераторы.

Для добавления использования итераторов нужно добавить вложенный класс итератора `iterator` и методы `begin` и `end`, которые возвращают итераторы на первый элемент и элемент, следующий за последним.

```
Dynarray<string> a = {"Cat", "Dog", "Nutria", "Echidna", "Turtle", "Coati"};
for (Dynarray<string>::iterator it = a.begin(); it != a.end(); ++it) {
        cout << *it << endl;
}
```

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.