Моя памятка по C++

Если ты это читаешь - значит пришло время писать код на плюсах. В данный момент я тоже не помню как это делать, но у меня (в отличие от тебя) есть время вспомнить, поэтому тут будут находиться все штуки, которые ты наверняка забыл, глупая голова. Также в данном документе я укажу сокровенные знания с поясов по плюсам, короче должно получиться хорошо, надеюсь, что я это всё не забросил.

С чем я столкнулся с самого начала

- Компилирование со всеми прибамбасами:

```
g++ -pedantic -Wall -Wextra -Wcast-align -Wcast-qual -Wctor-dtor-privacy -Wdisabled-optimization -Wformat=2 -Winit-self -Wlogical-op -Wmissing-declarations -Wmissing-include-dirs -Wnoexcept -Wold-style-cast -Woverloaded-virtual -Wredundant-decls -Wshadow -Wsign-promo -Wstrict-null-sentinel -Wstrict-overflow=5 -Wswitch-default -Wundef -Werror -Wno-unused test.cpp  -g -o a
```

- Библиотека ввода-вывода `#include <iostream>`, а не `#include <stdio.h>`, хотя и она тоже
- Строки это `std::string`, лежат в `#include <string>`
- Итерация по всем элементам вектора может выглядеть вот так:

```
for (auto current_elem_name : vector_name) //current_elem_name - копия элемента вектора
{
    std::cout << _current_elem_name << " ";
}
```

- Чтобы создать вектор, содержимое которого известно заранее, используются фигурные скобки:

```
std::vector<int> days_in_month = {31, 28, 31, 30, 31};
```

- Понижение регистра строк:

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;
int main(void)
{
    string  name = "waR IS peAcE";
    for (auto& c : name)
    {
        c = tolower(c);
    }
    cout << name;
    return 0;
}
/* Output:
 * war is peace
 */
```

- Вернуть пустой(ое) словарь/вектор/множество

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

vector<int> GetVect(bool is_empty)
{
    if (is_empty)
```

```
    {
        return {};
    }
    else
    {
        return {2, 4, 12};
    }
}
```

Первая неделя Белого Пояса

1. `std::cin >> a >> b >> c;` работает именно так, как тебе хочется.
2. Перебрать слово по буквам можно с помощью `str_prot.at(i)`, где `i` - индекс буквы в слове `str_prot`. Метод вернет букву, разумеется. . Вывод - строки всё ещё массив чаров, как и в Си.
3. Чтобы развернуть вектор можно использовать `std::reverse` из

```
#include <iterator>
#include <algorithm>
```

Пример:

```
vector<int> a;
reverse(a.begin(), v.end());
```

Точно так же разворачивается и строка.

Вторая неделя Белого Пояса

Векторы

1. Сортируем

```
#include <algorithm>
#include <vector>

using namespace std;
int main(void)
{
    vector<int> nums = {3, 6, 1, 2, 0, 2};
    sort(begin(nums), end(nums));
    return 0;
}
```

2. Константные ссылки

Существует проблема: иногда нам хочется вызвать функцию, которая не должна менять входные параметры, но при этом ей на вход могут подаваться очень большие структуры. Если передавать параметры "традиционным" способом, то произойдет полное (глубокое) копирование структуры, что может занимать место и память. Передача параметра по ссылке частично решит нашу проблему - копироваться будет меньшее количество данных

(очень малое), однако тогда функция сможет поменять исходные данные, которые были ей переданы. Чтобы полностью решить нашу проблему можно воспользоваться `const`:

```
void foo(const int& x)
```

Теперь копироваться будет лишь адрес, однако при попытке изменить содержимое параметра программа просто не станет компилироваться!

Также "константная ссылка" позволит написать следующий кусок кода, который не вызовет ошибок компиляции:

```
void foo(const int& x)
{
    ...
}
int bar()
{
    ...
    return ...;
}
int main(void)
{
    foo(bar); //не сохраняем ничего в отдельную переменную, а сразу отправляем результат в foo
    return 0;
}
```

Если в данном сниппете убрать `const`, то программа перестанет компилироваться, так как по правилам C++ результат вызова функции не может быть передан "по ссылке" в другую функцию.

3. На самом деле, `const` это специальный модификатор типов данных, запрещающий изменение объектов. Если попытаться изменить `const` переменную, то программа не будет компилироваться. Если сделать "константный контейнер", то `const` будет распространяться и на элементы этого контейнера:

```
...
int main(void)
{
    const vector<string> w = {"hello"};
    w[0][0] = 'H'; //Из-за этой строчки программа не будет компилироваться, хотя мы просто поменяли первую букву первого элемента контейнера w
    return 0;
}
```

4. Как расширить вектор? Ответ: `resize`:

```
...
vector<int> a(28, 0);//Создаем вектор из 28 элементов заполненный нулями
a.resize(31);//Теперь в векторе 31 элемент, содержимое вектора a не
изменилось
a.assign(100, 1);//А теперь вектор имеет размер 100 и весь заполнен
единицами!
```

5. `vector_name.clear()` удаляет все элементы, после чего размер (size) `vector_name` становится равным нулю.

Словари (map)

Объявление:

```
#include <map>
map<int, string> slovar;
slovar[1950] = "sqooba";
slovar[2019] = "booba";
slovar[1950] = "Jija";
```

Перебор элементов

```
for (auto item : slovar)
{
    cout << item.first << " : "; //выводим ключ
    cout << item.second << ";" << endl; //значение
}
//1950 : Jija;
//2019 : booba;
```

Важное свойство map - ключи в словаре автоматически **сортируются** и при выводе предложенным способом сначала напечатается ключ и значение с меньшим значением ключа (то есть 1950).

Размер словаря можно получить так же как мы делали это для вектора:

```
cout << slovar.size() << endl; // в данном случае программа выведет 2 ("sqooba"
перезатрется на "Jija")
```

Обращение по ключу происходит так же, как и в векторе или массиве:

```
cout << slovar[1950] << endl; // "Jija"
```

Однако мы можем случайно (или нет) обратиться по ключу, которого еще нет в словаре:

```
cout << slovar[12] << endl;
```

В этом случае в словаре автоматически создаётся элемент словаря "ключ-значение" с ключом 12 и с неким значением по умолчанию для данного типа (в нашем случае `string`, для которого значение по умолчанию есть пустое слово)

Удаление элемента map осуществляется с помощью метода `map_name.erase(key_name)`

Объявление map с заранее известными размерами :

```
map<string, int> m  = {{"one", 1}, {"two", 2}, {"three", 3}};
```

Подсчет числа вхождений элемента словаря с ключом `key` осуществляется с помощью метода `count(key)`:

```
...
map<string, int> a = {{"Alex", 20}, {"Andrew", 19}, {"Sergey", 26}};
cout << a.count("Alex");
...
/* Output:
 * ...
 * 1
 * ...
 */
```

Так как все ключи в словаре уникальны, то `count` может возвращать либо 0 либо 1.

Новые фичи для map

"Свежевведенные" возможности языка. Теперь итерироваться по словарю можно удобнее:

```
map<string, int> m;
for (const auto& [key, value] : m)
{
    cout << key << ": " << value << endl;
}
```

Множества (set)

Контейнер с говорящим именем. Объявление, добавление и печать:

```
#include <iostream>
#include <string>
#include <set>

using namespace std;
void PrintPeople(const set<string>& s);
void PrintPeople(const set<string>& s)
{
    cout << "Size: " << s.size() << endl;
    for (auto x : s)
    {
        cout << x << endl;
    }
}
```

```
int main(void)
{
    set<string> famous_people;            //множество строк
    famous_people.insert("Morgenshtern"); // Добавляем известных людей
    famous_people.insert("Slava Marlow");
    famous_people.insert("Slava Marlow"); //Добавление второго Славы Марлова
ничего не измени - он уже там есть
    PrintPeople(famous_people);           //Вывод: Morgenshtern \n Slava Marlow -
-- в алфавитном порядке
    famous_people.erase("Slava Marlow");  //Удалили Славу,теперь там только
Алишер Тагирович
    return 0;
}
```

В данном примере видно: элементы во множестве хранятся в единственном экземпляре в отсортированном порядке. Размер контейнера выводится как и прежде через `set_name.size()`. Удаление осуществляется через `set_name.erase(key)`. Посчитать количество вхождений элемента `key` в множество можно так же, как и в случае с `map`: `set_name.count(key)` -- так как в множестве дублей не бывает, `count` может вернуть либо 1 либо 0.

Объявление множество с заранее известным количеством элементов:

```
...
set<string> famous_people = {"Slava Marlow", "Oxxxymiron", "Morgenshtern",
"Oxxxymiron"}; // оптическая иллюзия - в множестве Оксимирон лишь в одном
экземпляре
set<string> other_famous_people = {"Slava Marlow", "Oxxxymiron", "Morgenshtern"};
cout << (famous_people == other_famous_people) << endl; //Выведет 1.
...
```

Создаем **множество по вектору**:

```
#include <iostream>
#include <string>
#include <vector>
#include <set>

using namespace std;
void PrintSet(const set<string>& s);
void PrintSet(const set<string>& s)
{
    cout << "Size: " << s.size() << endl;
    for (auto x : s)
    {
        cout << x << endl;
    }
}

int main(void)
{
    vector<string> test_vector = {"a", "b", "a"};
    set<string> test_set(begin(test_vector), end(test_vector));
    PrintSet(test_set);
    return 0;
}
/* Output:
```

```
 *Size: 2
 *a
 *b
 */
```

Третья неделя Белого Пояса

Некоторые алгоритмы

count

`count(range1, range2, elem)` возвращает количество вхождений элемента `elem` в контейнер в диапазоне от `range1` до `range2`, лежит в `<algorithm>` . Пример:

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;
int main(void)
{
    vector<int> a = {1, 1, 1, 1, 2, 4 };
    cout << "Count as func: " << count(begin(a), end(a), 1) << endl;
    return 0;
}
/* Output:
 *Count as func: 4
 */
```

count_if

Позволяет осуществить подсчет элементов по условию. На вход заместо элемента принимает функцию, которая должна возвращать значения типа `bool`, а принимать на вход элемент контейнера: `count(range1, range2, func)` Пример:

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;
bool Greater_2(int x);
bool Greater_2(int x)
{
    return (x > 2);
}

int main(void)
{
    vector<int> a = {1, 1, 1, 1, 2, 4};
    cout << "Count as func: " << count_if(begin(a), end(a), Greater_2) << endl;
//выведет все элементы большие по модулю чем 2 (в данном случае такой элемент
единственный - 4)
    return 0;
}
/* Output:
```

```
 *Count as func: 1
 */
```

Лямбда выражения

В примере выше для того, чтобы посчитать количество элементов вектора больших по модулю, чем 2, нам пришлось писать отдельную "узко специализирующуюся" функцию `bool Greater_2(int x)`, которая скорее всего нигде и никогда в программе использоваться не будет, однако, чтобы посмотреть, что эта однострочная функция делает, придётся листать код в самый верх, а затем возвращаться обратно. Это *неудобно*!

Решение нашей проблемы состоит в "написании функции налету". Продемонстрируем на примере выше:

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;
int main(void)
{
    vector<int> a = {1, 1, 1, 1, 2, 4};
    cout << "Count as func: " << count_if(begin(a), end(a), [](int i)
    {
        return (i > 2);
    }
    ) << endl; //выведет все элементы большие по модулю чем 2 (в данном случае
такой элемент единственный - 4)
    return 0;
}
/* Output:
 *Count as func: 1
 */
```

Как видно, мы объявили тело функции сразу как аргумент функции `count_if`, предварив её квадратными скобками `[]`. Эта функция принимает на вход `int i` - очередной элемент контейнера.

Квадратные скобки нужны, чтобы "показать" (иначе говоря, передать в контекст) лямбда-функции какие-то переменные, объявленные выше в коде, например:

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;
int main(void)
{
    vector<int> a = {1, 1, 1, 1, 2, 4};
    int thr;
    cin >> thr; //вводим порог для подсчета
    cout << "Count as func: " << count_if(begin(a), end(a), [thr](int i)
    {
        return (i > thr); //ошибки не будет ведь thr указана в квадратных скобках
    }
```

```
    ) << endl; //выведет все элементы большие по модулю чем 2 (в данном случае
такой элемент единственный - 4)
    return 0;
}
/*Input:
 *2
 *Output:
 *Count as func: 1
 */
```

Теперь мы можем задавать порог подсчета элементов извне!

Range-based for

Допустим, что задача состоит в увеличении всех элементов вектора на единицу. Как это можно сделать ***неправильно***:

```
vector<int> v = {1, 2, 3, 4};
for (auto x : v) //x - копия элемента в векторе v, а не сам элемент
{
    x++; //увеличиваем копию на 1
}//копия "затирается" здесь
```

Правильное решение:

```
vector<int> v = {1, 2, 3, 4};
for (auto& x : v) //x - копия ссылки на элемент в векторе v
{
    x++; //увеличиваем элемент вектора по ссылке на единицу
}//копия ссылки "затирается" здесь, но само значение в векторе уже изменено
```

Сортировка со своим компаратором

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;
int main(void)
{
    vector<int> v = {1, -3, 2};
    sort(begin(v), end(v), [](int a, int b) {return abs(a) < abs(b);}); //
сортировка по возрастанию модулей чисел с использованием лямбда выражения.
    //вектор v будет выглядеть вот так: {1, 2, -3}
    return 0;
}
```

Видимость и инициализация переменных

Скомпилируется ли следующий код?

```
#include <iostream>

using namespace std;
int main(void)
{
    string s = "Hello";
    {
        string s = "world!";
        cout << s << endl;
    }
    cout << s << endl;
    return 0;
}
```

Если не указан `-Wshadow` при компиляции, то код скомпилируется и выполнится с выводом:

```
world!
hello
```

В данном случае происходит так называемое "затенение" переменной `s`.

Введение в структуры и классы

Изменяем неизменяемое

Рассмотрим сниппет с, казалось бы, приватным полем `string name`;

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

class Human
{
public:
    string& GetName()
    {
        return name;
    }
    void SetName(string _name)
    {
        name = _name;
    }
private:
    string name;
};

int main(void)
{
    Human guy;
    guy.SetName("Denis");         //герой явно мужчина - имя Денис
    cout << guy.GetName() << endl;//Вывод: Denis\n
    guy.GetName() = "Kate";       //Поле name приватное - ничего у вас не выйдет!
Или нет!?
    cout << guy.GetName() << endl;//Вывод: Kate\n
    return 0;
}
```

Дело в том, что `GetName` возвращает нам *ссылку* на строку, по которой мы можем написать всё, что душе угодно!!

Методы в структурах

Удивительно, но факт. В структурах есть методы и они работаю так, как будто это класс, у которого всё `public`

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

struct Animal
{
    void Say(){ cout << voice << endl;};
    string voice;
};

int main(void)
{
    Animal cat, dog;
    cat.voice = "Meow";
    dog.voice = "Bark";
    cat.Say();          //Мяукнет
    dog.Say();          //Гавкнет
    return 0;
}
```

Количество `public` и `private`

Секции public и private в определении класса могут повторяться любое количество раз и располагаться в любом порядке.

Константность методов

Рассмотрим следующий кусок кода:

```
#include <iostream>

using namespace std;

class Human
{
    public:
    void SetAge(int _age)
    {
        age = _age;
    }
    void SetHeight(double _h)
    {
        height = _h;
    }
    void SetName(string _n)
    {
        name = _n;
```

```
    }
    int GetAge() const //константный метод -- то есть метод не меняет текущий
объект
    {
        return age;
    }
    double GetHeight() const //константный метод -- то есть метод не меняет
текущий объект
    {
        return height;
    }
    string GetName() const //константный метод -- то есть метод не меняет текущий
объект
    {
        return name;
    }
    private:
    int age;
    double height;
    string name;
};

void PrintHuman(const Human& man);
void PrintHuman(const Human& man)
{
    cout << "Name: " << man.GetName() << endl;  //так как методы константные, то
их вызов
    cout << "Age: " << man.GetAge() << endl;    //не изменит объект man
    cout << "Height: " << man.GetHeight() << endl;
}

int main(void)
{
    Human Alex;
    Alex.SetAge(10);
    Alex.SetHeight(150.5);
    Alex.SetName("Alex");
    PrintHuman(Alex);
    return 0;
}
```

Вся суть сниппета в использовании функции `void PrintHuman(const Human& man)`, легко заметить, что на вход она принимает *константную* ссылку на класс `Human`. Это крайне логично -- мы не хотим копировать лишний раз экземпляр класса, но и менять ничего не собираемся, однако компилятору это может быть и не очевидно.

Внимательный читатель обратит свой взор на странные методы в этом классе `int GetAge() const`, `double GetHeight() const` и `string GetName() const`. Эти `const` как раз и говорят компилятору, что методы никак не смогут изменить объект, то есть `const` в `void PrintHuman(const Human& man)` будет соблюдаться, если из неё вызвать эти методы.

Конструкторы

Знакомство

Конструктор - специальный метод класса, у которого нет возвращаемого значения, с названием совпадающим с названием класса.

Пример:

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

class Human
{
    public:
    Human(const string& _name) //Тот самый
    {
        name = _name;
    }
    void SetName(string _n)
    {
        name = _n;
    }
    string GetName() const
    {
        return name;
    }
    private:
    string name;
};

void PrintHuman(const Human& man);
void PrintHuman(const Human& man)
{
    cout << "Name: " << man.GetName() << endl;
}

int main(void)
{
    //Human Alex; - теперь так создать объект не получится - программа не
скомпилируется, но зато можно вот так:
    Human Alex("Alex");  //Создали объект класса Human сразу с именем "Alex"
    PrintHuman(Alex);
    return 0;
}
/*Output:
 *Name: Alex
 */
```

Обратите внимание, что в части "Константность методов" похожий объект создавался с помощью отдельных методов.

Важно: конструктор, добавленный в `private`, как и любой приватный метод, нельзя будет вызвать снаружи класса.

Конструкторы по умолчанию

Как видно в примере выше строка кода `Human Alex;` теперь будет провоцировать ошибку компиляции, это связано с тем, что отсутствует так называемый конструктор по умолчанию:

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

class Human
{
    public:
    Human()                    //Конструктор по умолчанию, раньше компилятор добавлял его в класс сам
    {
        name = "None";
    }
    Human(const string& _name) //Конструктор, требующий константную ссылку на строку в качестве параметра
    {
        name = _name;
    }
    void SetName(string _n)
    {
        name = _n;
    }
    string GetName() const
    {
        return name;
    }
    private:
    string name;
};

void PrintHuman(const Human& man);
void PrintHuman(const Human& man)
{
    cout << "Name: " << man.GetName() << endl;
}

int main(void)
{
    Human Alex;  //Создали объект класса Human с помощью конструктора по умолчанию
    PrintHuman(Alex);
    //Аналогично будут работать:
    //PrintHuman(Human());
    //PrintHuman({}); <-- тут тип понятен из заголовка функции void PrintHuman(const Human& man)
    //PrintHuman({"Alex"}); <-- выведет Name: Alex
    return 0;
}
/*Output:
 *Name: None
 */
```

Пример использования

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

class Human
{
    public:
    Human()
    {
        name = "None";
    }
    Human(const string& _name)
    {
        name = _name;
    }
    void SetName(string _n)
    {
        name = _n;
    }
    string GetName() const
    {
        return name;
    }
    private:
    string name;
};

void PrintHuman(const Human& man);
void PrintHuman(const Human& man)
{
    cout << "Name: " << man.GetName() << endl;
}

Human CreateHuman(bool is_nameless);
Human CreateHuman(bool is_nameless)
{
    if (is_nameless == true)
    {
        return {};
    }
    else
    {
        return {"Smith"};
    }
}
int main(void)
{
    PrintHuman(CreateHuman(true));
    return 0;
}
/*Output:
 *Name: None
 */
```

Функция `Human CreateHuman(bool is_nameless)` с помощью конструктора по умолчанию возвращает либо безымянного человека, либо очередного Смита.

Конструктор в структурах

Рассмотрим код ниже:

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

struct Human
{
    int age = 0;
    string name = "No name";
    string surname = "No surname";
    Human(int _age, string _name, string _surname)
    {
        age = _age;
        name = _name;
        surname = _surname;
    }
    Human() {}
};

int main(void)
{
    Human Alex(20, "Alex", "Ivanov");
    Human Unknown;
    cout << "Unknown's Surname : "<< Dron.surname << endl;
    cout << "Age: " << Alex.age << "\n" << "Name: " << Alex.name << "\n" <<
"Surname: " << Alex.surname << endl;
    Human Dima = {20, "Dima", "Ivanov"};
    cout << "Age: " << Dima.age << "\n" << "Name: " << Dima.name << "\n" <<
"Surname: " << Dima.surname << endl;
    return 0;
}
/*Output:
 *Unknown's Surname : No surname
 *Age: 20
 *Name: Alex
 *Surname: Ivanov
 *Age: 20
 *Name: Dima
 *Surname: Ivanov
 */
```

Деструкторы

Деструкторы - специальные методы, которые используются при уничтожении объекта. Назначение деструкторов - откат действий, сделанных в конструкторе и других методах:

- Закрытие открытого файла
- Освобождение выделенной вручную памяти
- Прочее (всё, что кажется полезным в зависимости от ситуации)

Пример:

```
#include <iostream>

using namespace std;
```

```
class Human {
  public:
    Human(const string& _name, int _age) {
      name = _name;
      age = _age;
    }
    ~Human() { //непосредственно сам деструктор класса Human
      cout << name << " of age " << age << " was destroyed" << endl;
    }
  private:
    string name;
    int age;
};

int main(void) {
  Human person("Alex", 20);
  return 0;
}// В данном месте person будет уничтожена компилятором, поэтому сработает наш деструктор:

/*Output:
 *Alex of age 20 was destroyed
 */
```

Время жизни объекта

Порядок уничтожения объектов

В каком порядке будут уничтожаться объекты в сниппете ниже?

```
#include <iostream>

using namespace std;

class Human {
  public:
    Human(const string& _name, int _age){
      name = _name;
      age = _age;
    }
    ~Human() {
      cout << name << " of age " << age << " was destroyed" << endl;
    }
  private:
    string name;
    int age;
};

int main(void) {
  Human person1("Alex", 20);
  Human person2("Max", 20);
  return 0;
}
```

Ответ, **в обратном порядке относительно создания**:

```
Max of age 20 was destroyed
Alex of age 20 was destroyed
```

Подобно стеку - последним пришёл, первым ушёл.

Уничтожение объекта, переданного в качестве параметра функции

```
#include <iostream>

using namespace std;

class Human {
  public:
    Human() {
      name = "Ivan";
      age = 18;
      cout << "Default constructed" << endl;
    }
    Human(const string& _name, int _age){
      name = _name;
      age = _age;
    }
    ~Human() {
        cout << name << " of age " << age << " was destroyed" << endl;
    }
  private:
    string name;
    int age;
};

void TestFun (const Human& man);
void TestFun (const Human& man) {
    cout << 2 << endl;;
}

int main(void) {
  cout << 1 << endl;
  TestFun({});
  cout << 3 << endl;
  return 0;
}
/*Output:
 *1
 *Default constructed
 *2
 *Ivan of age 18 was destroyed
 *3
 */
```

Вывод: объект, переданный продемонстрированным образом в функцию, будет удалён при выходе из этой функции, как и все её локальные переменные и остальные параметры.

Четвертая неделя Белого Пояса

Неявные преобразования

Рассмотрим сниппет, в котором написана структура `Date`. На первый взгляд, всё сделано так, чтобы пользующийся структурой всегда понимал, где в конструкторе `Date` день, где месяц, а где год. Однако не всё так гладко:

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

struct Day {
  int value;
  Day (int v) {         //Теперь компилятор умеет создавать из int тип Day
    value = v;
  }
};

struct Month {
  int value;
  Month (int v) {       //Теперь компилятор умеет создавать из int тип Month
    value = v;
  }
};

struct Year {
  int value;
  Year (int v) {        //Теперь компилятор умеет создавать из int тип Year
    value = v;
  }
};

struct Date {
  int day;
  int month;
  int year;
  Date(Day new_day, Month new_month, Year new_year) {
      day = new_day.value;
      month = new_month.value;
      year = new_year.value;
  }
};

void PrintDate(const Date& d);
void PrintDate(const Date& d) {
  cout << d.day << "." << d.month << "." << d.year << endl;
}

int main(void) {
  Date new_date = {10, 11, 12}; //Что из этого день, месяц и год? Почему это
компилируется?
  PrintDate(new_date);
  return 0;
}
```

Добавив в структуры `Day`, `Month` и `Year` конструкторы, мы научили программу приводить тип `int` к соответствующему типу, поэтому в строке `Date new_date = {10, 11, 12};` **неявное приведение типов!** Компилятор автоматически создает из `10` тип `Day`, из `11` - `Month` и из `12` - `Year`.

Как запретить компилятору неявно преобразовывать тип?

Добавим слово `explicit`, что значит "явный", в объявлении конструкторов в структурах:

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

struct Day {
  int value;
  explicit Day (int v) {        //Теперь компилятор умеет создавать из int тип
Day, но только явно
    value = v;
  }
};

struct Month {
  int value;
  explicit Month (int v) {      //Теперь компилятор умеет создавать из int тип
Month, но только явно
    value = v;
  }
};

struct Year {
  int value;
  explicit Year (int v) {       //Теперь компилятор умеет создавать из int тип
Year, но только явно
    value = v;
  }
};
```

Теперь это `Date new_date = {10, 11, 12};` и даже это `Date new_date = {{10}, {11}, {12}};` не сработает, поэтому придётся делать это явно, что улучшит читабельность:

```
...
  Date new_date = {Day(10), Month(11), Year(12)};
  //Либо вот так: Date new_date = {Day{10}, Month{11}, Year{12}};
...
```

Универсализация кода с помощью классов на примере класса функций

Постановка проблемы

Пусть имеется некая программа для работы с картинками, которые описываются структурой `Image`:

```
struct Image {
  double quality;
  double freshness;
};
```

Для работы с картинками определяются две функции:

```
struct Params {
  double a;
  double b;
}; //Вспомогательная структура для вычисления

//Вычисление веса происходит по формуле: weight = quality - freshness * a + b
double ComputeImageWeight (const Params& params, const Image& image) {
  double weight = image.quality;
  weight -= image.freshness * params.a + params.b;
  return weight;
}

//Существует также и обратная функция, которая по весу сообщает свежесть
double ComputeQualityByWeight(const Params& params, const Image& image, double
weight) {
  double quality = weight;
  quality += image.freshness * params.a + params.b;
  return quality;
}
```

И вот нам говорят, что в структуру `Image` необходимо добавить ещё одно поле `double rating`, и всё бы ничего, однако `rating` участвует в формуле вычисления веса, поэтому необходимые изменения примут вид:

```
struct Image {
  double quality;
  double freshness;
  double rating;
};

struct Params {
  double a;
  double b;
  double c; // константа для учёта rating в формуле
};

//Вычисление веса происходит по формуле: weight = quality - freshness * a + b +
rating * c
double ComputeImageWeight (const Params& params, const Image& image) {
  double weight = image.quality;
  weight -= image.freshness * params.a + params.b;
  weight += image.rating * params.c;
  return weight;
}

//Изменения коснутся и второй фуникци
double ComputeQualityByWeight(const Params& params, const Image& image, double
weight) {
```

```
  double quality = weight;
  quality += image.freshness * params.a + params.b;
  quality -= image.rating * params.c;
  return quality;
}
```

-Так в чём же проблема? Поменяли структуру - поменяем и функции.

Действительно, в данном примере всё выглядит цивильно, однако, если функции больше, то можно случайно забыть поменять одну из них и тогда, всё станет в лучшем случае **очень плохо**. Существует назойливая проблема - проблема модификации. Помимо неё - неявное дублирование кода.

Решение проблемы

Создадим класс `Function`, который будет олицетворять функцию вычисления нашей формулы, и функцию `Function MakeWeightFunction(const Params& params, const Image& image)`, возвращающую нам нужный объект класса `Function`. Тогда функции `ComputeImageWeight` и `ComputeQualityByWeight` примут вид:

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

struct Image {
  double quality;
  double freshness;
  double rating;
};

struct Params {
  double a;
  double b;
  double c; // константа для учёта rating в формуле
};

//Вспомогательный класс, объекты которого содержат пары "операция - текущее
значение"
class FunctionParts {
  public:
    FunctionParts(char new_operation, double new_value)  {
      operation = new_operation;
      value = new_value;
    }
    double Apply(double source_value) const {
      if (operation == '+') {
        return source_value + value;
      } else {
        return source_value - value;
      }
    }
    void Invert() {
      if (operation == '+') {
        operation = '-';
      } else {
```

```
        operation = '+';
      }
    }
  private:
    char operation;
    double value;
};

class Function {
  public:
    double Apply(double value) const {
      for (const FunctionParts& part : parts) {
        value = part.Apply(value);
      }
      return value;
    }
    void Invert() {
      for (FunctionParts& part : parts) {
        part.Invert();
      }
      reverse(begin(parts), end(parts));
    }
    void AddPart(char operation, double value) {
      parts.push_back(FunctionParts(operation, value));
    }
  private:
    vector<FunctionParts> parts;
};

Function MakeWeightFunction(const Params& params, const Image& image) {
  Function function;
  //Функция как-то создает обхект по паметрам
  function.AddPart('-', image.freshness * params.a + params.b);
  function.AddPart('+', image.rating * params.c);
  return function;
}

//Вычисление веса происходит по формуле: weight = quality - freshness * a + b +
rating * c
double ComputeImageWeight (const Params& params, const Image& image) {
  Function function = MakeWeightFunction(params, image);
  return function.Apply(image.quality);
}

//Изменения коснутся и второй фуникци
double ComputeQualityByWeight(const Params& params, const Image& image, double
weight) {
  Function function = MakeWeightFunction(params, image);
  //Обращаем полученную функцию
  function.Invert();
  return function.Apply(weight);
}
```

Теперь вся работа с формулой состоит в обращении к `Function function`, а менять формулу можно из одного места - `MakeWeightFunction`.

Работа с текстовыми файлами и потоками

Базовые классы

- `istream` - поток ввода (cin)
- `ostream` - поток вывода (cout)
- `iostream` - поток ввода-вывода

Потоки для работы с файлами

`#include <fstream>`:

- `ifstream` - для чтения (наследуется от `istream`)
- `ofstream` - для записи (наследуется от `ostream`)
- `fstream` - для чтения и записи (наследуется от `iostream`)

Пример чтения из файла

`getline`

Пусть есть файл с названием `TestFile.txt` и содержимым:

```
Hello world!
second line
```

Тогда считать его можно с помощью функции `getline` следующим образом:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;
int main(void) {
    //Так как мы планируем читать из файла, то переменная input будет иметь тип
ifstream
    //Конструктор ifstream в качестве аргумента принмиает путь до файла
    ifstream input("TestFile.txt");
    string line;
    getline(input, line);
    cout << line << endl;
    getline(input, line);
    cout << line << endl;
    return 0;
}
/*Output:
 *Hello world!
 *second line
 */
```

`getline` **первым аргументом** принимает **поток**, откуда мы собираемся читать, а **вторым строку**, в которую мы планируем поместить прочитанное. Исходя из названия, данная **функция считывает символы в строку до первого** `'\n'`

Замечание: если в примере выше продублировать строчки `getline(input, line); cout << line << endl;` ещё один раз, то вывод программы будет таким:

```
/*Output:
 *Hello world!
 *second line
 *second line
 */
```

Объяснение: в данном случае `getline` уже в третий раз не срабатывает так, как нужно и оставляет переменную `line` неизменной!

TODO: продолжить конспект с 3:52.