Моя памятка по C++

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.

Общие факты

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
```

Компилирование с `C++17`

Для компилирования с `C++17` нужно добавить к коду выше фразу `-std=c++17`

Библиотека ввода-вывода

`#include <iostream>`, а не `#include <stdio.h>`, хотя и она тоже

Что такое строки?

Строки это `std::string`, лежат в `#include <string>`

Итерация по всем элементам вектора

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

Вектор с фиксированным количеством элементов

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};
    }
}
```

Проверка на конец потока

```
{
    ...
    if(stream.eof())
    {
        ...
    }
}
```

Удобная запись больших числовых констант

В `C++` есть возможность разбивать разряды в числе одинарной кавычкой `'` для лучшей читаемости:

```
{
    int x = 2'000'000'000; //Два миллиарда
}
```

Сделать число беззнаковым

Сделать число беззнаковым можно следующим образом:

```
{
    ...
    cout << (-1 < 1u) << endl; // -1 -> int, 1u -> unsigned int
    ...
}
//Если убрать флаг -Werror, предупреждающие о потенациально нежелаемом поведении, вывод:
/*Output:
 *0
 */
```

P.S. Вывод такой, потому что `-1` приведется к типу `unsigned int` и станет большим числом. (*см. Первую Неделю Желтого Пояса*)

Приведение типов

Приведение типов обычно осуществялется с помощью `static_cast`:

```
{
    ...
    int sum;
    vector<int> t = {1, 2, 3};
    cout << sum / static_cast<int>(t.size()) << endl; //t.size() имеет тип size_t
    ...
}
```

Переменная в `switch-case`

Чтобы определить переменную в кейсе свитч-конструкции **необходимы** фигурные скобки:

```
enum class RequestType {
  ADD,
  REMOVE,
  NEGATE
};

void ProcessRequest(
    set<int>& numbers,
    RequestType request_type,
    int request_data) {
  switch (request_type) {
  case RequestType::ADD:
    numbers.insert(request_data);
    break;
  case RequestType::REMOVE:
    numbers.erase(request_data);
    break;
  case RequestType::NEGATE: {  // фигурные скобки обязательны
    bool contains = numbers.count(request_data) == 1;
    if (contains) {
      numbers.erase(request_data);
      numbers.insert(-request_data);
    }
    break;
  }
  default:
    cout << "Unknown request" << endl;
  }
}
```

Интересный пример переполнения

Очевидное неприятное

```
{
    ...
    uint64_t sum = 0;
    int a, b, c;
    cin >> a >> b >> c;  //Входне данные влезают в размер int'a
    sum += a * b * c;    //А вот (a*b*c) не влезает в размер int'a, а приводится
именно к нему :(
    cout << sum << endl; //Видим переполнение :(
}
```

Почти хороший способ сравнивать даты

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

using namespace std;

struct Date
{
  int year;
  int month;
  int day;
};

bool operator< (const Date& lhs, const Date& rhs) {
  return vector<int>{lhs.year, lhs.month, lhs.day} < vector<int>{rhs.year,
rhs.month, rhs.day};
}

int main(void) {
  cout << (Date{2021, 12, 31} < Date{2001, 9, 7}) << endl;
  return 0;
}
/*Output:
 *0
 */
```

Хороший способ сравнивать даты

```
#include <iostream>
#include <tuple>

using namespace std;

struct Date
{
  int year;
  int month;
  int day;
};

tuple<const int&, const string&, const int&> GetRank(const Date& date) {
  return tie(date.year, date.month, date.day);
}
```

```
bool operator<(const Date& lhs, const Date& rhs) {
  return GetRank(lhs) < GetRank(rhs);
}
```

Объявление клички для сложного типа

Предотвращение дублирования кода с помощью объявления клички какому-то типу:

```
#include "beautiful output.h"

using Dictionary = map<char, set<string>>;

int main(void) {
    Dictionary a;
    a['c'].insert("creator");
    a['c'].insert("crab");
    a['b'].insert("brain");
    cout << a << endl;
    return 0;
}
/*Output
 *{(b,<brain>),(c,<crab,creator>)}
 */
//P.S. beautiful output.h можно посмотреть в первой неделе жёлтого пояса в
разделе "Универсальные функции вывода контейнеров в поток"
```

Количество объявлений и определений

Функция можеты быть ОПРЕДЕЛЕНА только один раз
Функция может быть ОБЪЯВЛЕНА несколько раз

Объявление и определение классов

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

//Объявление класса Human
class Human {
private:
  size_t age;
  string name;
public:
  Human(size_t _age, string _name);

  size_t GetAge() const;

  const string& GetName() const;
};

//Определение методов класса
Human::Human(size_t _age, string _name) {
  age = _age;
  name = _name;
}

size_t Human::GetAge() const {
```

```
  return age;
}

const string& Human::GetName() const {
  return name;
}
```

Компиляция и её этапы

препроцессинг (#include, #define) → компиляция всех `.cpp` файлов в объектные файлы → компоновка (линковка) объектных файлов → ELF

Отличие массива от указателя

Рассмотрим:

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

int main(void) {
  int a[100];
  int *p = a;
  return 0;
}
```

Вопрос: **в чём отличие между `p` и `a`?**

Оказывается, так как `a` это статически объявленный массив, то он находится в стеке программы и имеет отдельный тип `int[100]`, который в строке `int *p = a` приводится к `int *`. Однако, если `int[100]` **можно** привести к `int *`, то `int *` **нельзя привести** к `int[100]`.

На этом отличия не заканчиваются:

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

int main(void) {
  int a[100];
  int *p = a;
  cout << sizeof(a) << " " << sizeof(p) << endl;
  return 0;
}
/*Output
 *400 8
 */
```

`sizeof(a)` вывел 400 байт, что логично - в стеке выделено 100 ячеек по 4 байта для каждого инта, а `sizeof(p)` выводит 8, потому что размер указателя всегда 8 байт.

Более того, для статически объявленных массивов нет операции `=` :

```
using namespace std;

int main(void) {
  int a[100];
  int b[100];
  int x = 0;
  int *p = &x;
  a = b; // В этом месте будет ошибка
  a = p; // И в этом тоже
  return 0;
}
```

А вот для указателей есть:

```
using namespace std;

int main(void) {
  int x;
  int *p = &x;
  int *pp;
  pp = p; //Всё хорошо и p и pp указатели на x
  return 0;
}
```

Квадратные скобки как синтаксический сахар

Оказывается, что операция `a[i]` полностью равносильна `*(a + i)`, где `a` - имя массива:

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

int main(void) {
  int a[100];
  a[2] = 12; //<=> *(a + 2) = 12;
  cout << *(a + 2) << endl;
  *(a + 2) = 13;
  cout << a[2] << endl;
}
/*Output
 *12
 *13
 */
```

Поэтому работают вот такие страшные конструкции:

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

int main(void) {
  int a[100];
  5[a] = 42;   // <=> *(5 + a) = 42;
  cout << a[5] << endl;
}
/*Output
 *42
 */
```

Указатель на функцию

```
void f(int, int);
void f(int, int) {}

int main() {
  void(*pf)(int, int) = &f;
  //Тип у переменной pf - void(*)(int, int)
  return 0;
}
```

Выбор функции компилятором

```
#include <iostream>
using namespace std;
double f(double);
int f(int);

double f(double) {cout << 1 << endl; return 0;}
int f(int) {cout << 2 << endl; return 0;}

int main() {
  int x = f(0.0); //Вызывается double f(double) несмотря на то что x имеет тип
int
  return 0;
}
/*Output:
 *1
 */
```

Вывод: возвращаемый тип не влияет на выбор функции - влияют аргументы

Что НЕЛЬЗЯ делать со ссылками?

Во-первых, не получится создать ссылку на ссылку:

```
int x;
int&& r = x;
```

Самое коварное, что это может даже скомпилироваться, поскольку обозначение зарезервировано другим функционалом языка.

Во-вторых, нельзя объявлять ссылку без их инициализации:

```
int& r;
```

Это уже вызовет ошибку компиляции. **Причем** инициализировать ссылки можно только `lvalue` выражениями, а не `rvalue` выражениями, то есть `int& r = 1` не скомпилируется.

В-третьих, после создании ссылки на некий объект нельзя будет этот объект поменять, то есть ссылка до скончания веков будет относиться к объекту, с помощью которого она была создана

В-четверых, не получится создать вектор (или другой контейнер) из ссылок:

```
vector<int&> v; //Ошибка компиляции
```

Инициализация константных ссылок

Как мы поняли из написанного выше - нельзя инициализировать ссылки `rvalue`, однако в случае константных ссылок дело обстоит иначе:

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

int main() {
  const std::string& s = "abc";  //если бы не было const, то программа бы не
скомпилировалась
  std::cout << s << "\n";
  return 0;
}
```

Это сделано для того, чтобы можно было передавать `rvalue` в качестве параметра в функцию, которая принимает на вход константную ссылку:

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

// функция поиска образца sample в тексте text
bool foo(const std::string& text, const std::string sample) {
  // ...
  return true;
}

int main() {
  std::string text = "Hello world!";
  std::cout << foo(text, "world") << "\n";
  return 0;
}
```

При этом работает это так: неявно создается вспомогательный объект (в данном случае строка), который будет уничтожен, когда область видимости ссылки закончится, например:

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

int main() {
  {
    const std::string& text = "Hello world!"; // В данном месте неявно создается
строка, она не уничтожается, пока
    //объект с именем text не выйдет из области видимости
  }// В данном месте уничтожается text, а вместе с ним и вспомогательная строка
  return 0;
}
```

В связи с этим можно словить `undefined behavior` в виде `dangling reference`, то есть получить ссылку на объект, которого уже не существует. Пример:

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

const std::string& foo(const std::string& s);
const std::string& foo(const std::string& s) {
  return s;
}

int main() {
  std::cout << foo("bca") << "\n";
  return 0;
}
```

Что за ужас тут творится?

При вызове `foo` от `rvalue` создаётся вспомогательная строка, с которой ассоциируется ссылка, переданная в аргумент функции. Далее, в самой функции, эта ссылка возвращается в `main`, а затем **область её видимости заканчивается** (поскольку она была создана при вызове `foo`), а значит и вспомогательная строка **уничтожается**. В результате мы получаем ссылку на объект, который больше не существует (`dangling reference` по определению).

Константные указатели

Рассмотрим на примере:

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

int main() {
  int x = 12;
  const int *p = &x; //В данном случае const распространяется на то же, на что
ссылается указатель
  (*p)++; //<- Ошибка компиляции
  p++;    //Разрешено
  return 0;
}
```

А что если, мы хотим завести указатель, который изменять нельзя, а переменную можно? Тогда делаем так:

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

int main() {
  int x = 12;
  //Поменяем const и int* местами
  int* const p = &x; //В данном случае const распространяется на то же, на что
ссылается указатель
  (*p)++; //Разрешено
  p++;    //<- Ошибка компиляции
  return 0;
}
```

А что если, мы хотим завести неизменяемый указатель на неизменяемую переменную? Тогда делаем вот так:

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

int main() {
  int x = 12;
  const int* const p = &x; //В данном случае const распространяется на то же, на
что ссылается указатель
  (*p)++; //<- Ошибка компиляции
  p++;    //<- Ошибка компиляции
  return 0;
}
```

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(), a.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'` (`\n` при этом в строку не входит).

А что будет, если считать строку ещё раз?

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!

А как же тогда понять, что считывать больше ненужно?

Оказывается `getline` возвращает ссылку на поток, из которого она читает строчки. Ссылку на поток можно преобразовать в `bool`: `true`, если файл ещё не кончился и можно считать следующую строчку и `false` иначе. Тогда **готовый код** примет вид:

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

using namespace std;
int main(void) {
  ifstream input("TestFile.txt");
  string line;
  while (getline(input, line)) {
    cout << line << endl;
  }
  return 0;
}
/*Output:
 *Hello world!
 *second line
 */
```

Аккуратное создание потока

Что если мы ошиблись в названии файла?

Чтобы программа сказала нам, что она не может открыть файл, наш код нужно подкорректировать:

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

using namespace std;
int main(void) {
  ifstream input("TestFile1.txt");
```

```
  string line;
  if (input.is_open() == false) {    //этот код можно упростить до if (!input)
    cout << "File open fail" << endl;
    return 0;
  }
  while (getline(input, line)) {
    cout << line << endl;
  }
  return 0;
}
```

Чтение данных через разделитель

Пусть нам нужно прочитать дату в формате 'year-month-day'. `TestFile.txt`:

```
2017-01-25
```

Тогда считать необходимое можно так:

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

using namespace std;
int main(void) {
  string file_name = "TestFile.txt";
  ifstream input(file_name);
  if (!input) {
    cout << "Error! Can not open a file with name: " << file_name << endl;
  }
  string year, month, day;
  getline(input, year, '-');
  getline(input, month, '-');
  getline(input, day, '-');
  cout << "Year: " << year << " Month: " << month << " Day: " << day << endl;
  return 0;
}
/*Output:
 *Year: 2017 Month: 01 Day: 25
 */
```

Чтение через операции ввода-вывода

Будем считывать то же самое из того же файла, что и выше, только другим способом:

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

using namespace std;
int main(void) {
  string file_name = "TestFile.txt";
  ifstream input(file_name);
  if (!input) {
    cout << "Error! Can not open a file with name: " << file_name << endl;
  }
```

```
  int year, month, day;
  year = 0;
  month = 0;
  day = 0;
  input >> year;   //считали 2017
  input.ignore(1); //проигнорировали -
  input >> month;  //счиатли 01
  input.ignore(1); //проигнорировали -
  input >> day;    //считали 25
  cout << "Year: " << year << " Month: " << month << " Day: " << day << endl;
  return 0;
}
/*Output:
 *Year: 2017 Month: 01 Day: 25
 */
```

Если же заранее неизвестно сколько в файле например чисел, то считать их можно так:

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

using namespace std;

int main() {
    ifstream input("input.txt");
    double a;
    while (input >> a) {
        cout << a << endl;
    }

    return 0;
}
```

Запись в файл

Запишем в файл `TestFile.txt` некую фразу.

Запись с удалением

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

using namespace std;
void ReadAll(const string& file_name);
void ReadAll(const string& file_name) {
  ifstream input(file_name);
  if (input) {
    string line;
    while (getline(input, line)) {
      cout << line << endl;
    }
  } else {
```

```
    cout << "ReadAll: cannot open file named " << file_name << endl;
  }
}

int main(void) {
  string file_name = "TestFile.txt";
  ofstream output(file_name);         //Содержимое файла удаляется и
перезаписывается
  if (output) {
    output << "Hello world!" << endl; //если в данной строке не выводить endl то
ReadAll ничего не прочтёт
  } else {
    cout << "Error! Can not open the file " << file_name << endl;
  }
  ReadAll(file_name);
  return 0;
}
```

Запись с дополнением

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

using namespace std;
void ReadAll(const string& file_name);
void ReadAll(const string& file_name) {
  ifstream input(file_name);
  if (input) {
    string line;
    while (getline(input, line)) {
      cout << line << endl;
    }
  } else {
    cout << "ReadAll: cannot open file named " << file_name << endl;
  }
}

int main(void) {
  string file_name = "TestFile.txt";
  ofstream output(file_name, ios::app); //открываем файл на дописывание в конец
  if (output) {
    output << "Hello world!" << endl;   //если в данной строке не выводить endl
то ReadAll ничего не прочтёт
  } else {
    cout << "Error! Can not open the file " << file_name << endl;
  }
  ReadAll(file_name);
  return 0;
}
```

Потоковые манипуляторы или форматирование вывода

```
#include <iomanip>

cout << fixed;           //для вывода double с фиксированной точностью (не черех экспоненту, а через точку)

cout << setprecision(2); //количество знаков после запятой теперь 2

cout << setw(10);        //задаем ширину вывода в символах для следующего вывода (сбрасывается после каждого вывода)

/*пример:
*no setw: [42]
*setw(6): [    42]
*/

cout << setfill('.')    //заполняем всё пустое пространство в выводе точкой
/*пример:
*no setw: [42]
*setw(6): [....42] <--- в ввыводе 6 символов
*/

cout << left           //выводим что-то слева в окошке с width
/*пример:
*no setw: [42]
*setw(6): [42....] <--- в ввыводе 6 символов
*/
```

Перегрузка операторов ввода и вывода

Рассмотрим уже привычную нам конструкцию:

```
cout << "hello " << "world";
```

Что на самом деле происходит в этой строчке?

Оказывается, что заместо `<<` вызывается функция `ostream& operator<<(ostream& stream, const string& s)`, а точнее `operator<<(cout, "hello ")`, и `hello` выводится на экран. Затем `operator<<` возвращает поток `cout` и вновь вызывается `operator<<(cout, "world")`. Поэтому код в рамке выше эквивалентен следующему:

```
operator<<(operator<< (cout, "hello "), "world");
```

Теперь, понимая как работает распечатка чего-либо в цепочке `smth << ... << smth << smth;`, мы можем написать свою логику взаимодействия функции `operator<<` (или `operator>>`) уже с нашими типами данных. Например:

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

using namespace std;

struct Human {
```

```
  string name = "Alex";
  int age = 24;
};

ostream& operator<<(ostream& stream, const Human& guy) {
  stream << guy.name << " " << guy.age;
  return stream;
}

istream& operator>>(istream& stream, Human& guy) {
    stream >> guy.name >> guy.age;
    return stream;
}

int main(void) {
  Human guy1;
  Human Ivan {"Ivan", 23};
  cout << guy1 << endl;
  cout << Ivan << endl;
  return 0;
}
/*Input:
 *Dima 19
 */

/*Output:
 *Alex 24
 *Ivan 23
 *Dima 19
 */
```

Грамотная перегрузка операторов ввода и вывода

Рассмотрим класс рациональных чисел:

```
#include <iostream>
#include <map>
#include <set>
#include <vector>
#include <numeric>
using namespace std;

class Rational {
public:
    Rational() {
        num = 0;
        denom = 1;
    }

    Rational(int numerator, int denominator) {
        const int g_c_d = gcd(numerator, denominator);
        num = numerator / g_c_d;
        denom = denominator / g_c_d;
        if (denom < 0) {
            denom = -denominator;
            num = -numerator;
        }
```

```
    }

    int Numerator() const {
        return num;
    }

    int Denominator() const {
        return denom;
    }

private:
    int num;
    int denom;
};
ostream& operator<<(ostream& stream, const Rational& r);
ostream& operator<<(ostream& stream, const Rational& r) {
    stream << r.Numerator() << "/" << r.Denominator();
    return stream;
}

istream& operator>>(istream& stream, Rational& r);
istream& operator>> (istream& is, Rational& r) {
  int n, d;
  char c;

  if (is) {
      is >> n >> c >> d;
      if (is) {
          if (c == '/') {
              r = Rational(n, d);
          }
          else {
              is.setstate(ios_base::failbit);
          }
      }
  }
}
```

Почему это <грамотно>?

```
...
istringstream s{"5*9"};
Rational r;

if (!(s >> r)) {
  // Ожидаемо, мы должны попасть сюда из-за некорректного разделителя у 5*9...
} else {
  // ...но мы попадем сюда, если не будем использовать особый метод setstate и
  // флаг ios_base::failbit
}
...
```

Перегрузка операторов + <

Обозначения: lhs => *left hand side*, rhs => *right hand side*

Научим программу складывать наши типы, а так же сравнивать их:

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

using namespace std;

struct Duration {
  int hour;
  int min;
  Duration(int _hour = 0, int _min = 0) {
    int total = _hour * 60 + _min;
    hour = total / 60;
    min = total % 60;
  }
};

ostream& operator<<(ostream& stream, const Duration& d) {
  stream << setfill('0');
  stream << setw(2) << d.hour << ":" << setw(2) << d.min ;
  return stream;
}

istream& operator>>(istream& stream, Duration& d) {
    stream >> d.hour;
    stream.ignore();
    stream >> d.min;
    return stream;
}

Duration operator+(const Duration& lhs, const Duration& rhs) {
  return Duration(lhs.hour + rhs.hour, lhs.min + rhs.min);
}

bool operator<(const Duration& lhs, const Duration& rhs) {
    if (lhs.hour == rhs.hour) {
      return lhs.min < rhs.min;
    } else {
      return lhs.hour < rhs.hour;
    }
}

void PrintVector(const vector<Duration>& v) {
    for (const Duration& e : v) {
        cout << e << " ";
    }
}


int main(void) {
```

```
  Duration test(0, 35);
  Duration test2(1, 25);
  Duration test3 = test + test2;
  cout << test3 << endl;
  vector<Duration> v = {test2, test, test3};
  PrintVector(v);
  cout << "\n";
  sort(begin(v), end(v));
  PrintVector(v);
  cout << "\n";
  return 0;
}
/*Output:
*02:00
*01:25 00:35 02:00
*00:35 01:25 02:00
*/
```

Введение в исключения

Исключения - специальный механизм языка `C++`, позволяющий отлавливать "неправильное" поведение кода и как-то его отлаживать. Рассмотрим пример:

```
#include <sstream>
#include <iostream>
#include <string>
#include <iomanip>

using namespace std;

struct Date {
  int year;
  int month;
  int day;
};

void EnsureNextSymbAndSkip(stringstream& stream);
void EnsureNextSymbAndSkip(stringstream& stream) {
  //Проверка следующего элемента
  if (stream.peek() != '/') {
    //Выкидываем исключение, если следующий символ не /
    stringstream ss;
    //Записываем причину ошибки в строчный поток, а затем выкидываем исключение в
what() которого запишется эта строка:
    ss << "Expected / but got: " << char(stream.peek());
    throw runtime_error(ss.str());
  }
  //Если всё хорошо то пропускаем /
  stream.ignore();
}

Date ParseDate(const string& str);
Date ParseDate(const string& str) {
  stringstream stream(str);
  Date date;
  stream >> date.year;
  //Игнорируем разделитель если он /
```

```
  EnsureNextSymbAndSkip(stream);
  stream >> date.month;
  //Игнорируем разделитель если он /
  EnsureNextSymbAndSkip(stream);
  stream >> date.day;
  return date;
}

int main(void) {
  string date_str("2017a01/25");
  try
  {
    //Здесь находится код, который потенциально может выкинуть исключение
    //Если исключения нет, то выполнится код в этом бллке
    Date date = ParseDate(date_str);
    cout << setw(2) << setfill('0') << date.day << '.';
    cout << setw(2) << setfill('0') << date.month << '.';
    cout << date.year;
  }
  catch(const exception& e)
  {
    //Если код выше выкинул исключение, то выполнится код в этом блоке
    //Выводим причину ошибки
    std::cerr << e.what() << '\n';
  }

  return 0;
}
/*Output:
 *Expected / but got: a
 */
```

Первая Неделя Жёлтого Пояса

Введение в целочисленные типы

Общие положения

```
#include <iostream>
#include <vector>
#include <typeinfo>


using namespace std;

int main(void) {
  vector<int> t = {-8, -7, 3};
  int sum = 0;                    //sum знаковое
  for (size_t x : t) {
      sum += x;
  }
  cout << sum/t.size() << endl;  //t.size() беззнаковое -> sum/t.size()
беззнаковое
  return 0;
}
```

Что выведет данный код?

Оказывается - `undefined behavior`, хоть код "логически" и верен. Дело в том, что `t.size()` возвращает число, которое не может быть отрицательным (на самом деле она возвращает целочисленный тип `size_t`). Затем при делении `int` на этот тип, `int` приводится к `size_t`, но `sum` имеет отрицательно значение `-12`, которое не может "правильно" привестись к `size_t`.

Целочисленные типы:

Классика:

- `int` - стандартный целочисленный тип
    1. `auto x` $\rightarrow$ `x` имеет тип `int`
    2. Наиболее эффективен, из-за чего на разных архитектурах может иметь разный размер
    3. Как правило, имеет размер 4 байта (32 бита)
- `unsigned int` - беззнаковый аналог `int`
    1. Как правило, имеет размер 4 байта
- `size_t` - тип представления размеров (`unsigned`)
    1. Результат вызова `size()` для контейнера
    2. 4 байта (на 32-ух битных) и 8 байт на (на 64-ех битных)

Типы с известным размером:

`#include <cstdint>`

- `int32_t` - знаковый, всегда 32 бита
- `uint32_t` - беззнаковый, всегда 32 бита
- `int8_t` и `uint8_t` - 8 бит
- `int16_t` и `uint16_t` - 16 бит
- `int64_t` и `uint64_t` - 64 бит

Просмотр минимального и максимального возможного значения типа

```
{
    ...
    cout << numeric_limts<int>::min() << endl;
    cout << numeric_limts<int>::max() << endl;
    ...
}
```

Преобразования целочисленных типов

Правила выведения общего типа

1. Перед сравнениями и арифметическими операциями числа приводятся к **общему типу**
2. Все типы размера **меньше** `int` приводятся к `int`
3. Из двух типов выбирается **больший** по размеру
4. Если размер одинаковый, выбирается **беззнаковый**

Примеры:

- `int / size_t` $\rightarrow$ `size_t` (см. третье правило)
- `int32_t + int8_t` $\rightarrow$ `int32_t` (см. третье правило)
- `int8_t * uint8_t` $\rightarrow$ `int` (см. втрое правило)
- `int32_t < uint32_t` $\rightarrow$ `uint32_t` (см. четвертое правило)

Правильная итерация по вектору

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

using namespace std;

int main(void) {
  vector<int> t = {5, 4};
  //for (int i = 0; i < t.size(); i++) не скомпилируется с -Werror из-за
приведения типа int к size_t
  //Первый способ
  for (size_t i = 0; i < t.size(); i++) {   //i и t.size() одинакового типа
size_t
      cout << t[i] << " ";
  }
  //Способ ниже подойдет, если t.size() "влезет" в int
  //Второй способ
  cout << endl;
  for (int i = 0; i < static_cast<int>(t.size()); i++) { //i и int(t.size())
одинакового типа int
      cout << t[i] << " ";
  }
  cout << endl;
  return 0;
}
```

Enum'ы и его маленькие радости

Рассмотрим их на примере работы обработки запросов. `void ProcessRequest(...)` принимает на вход ссылку на множество неких чисел, с которыми мы работаем, `ENUM` - тип запроса и некое третье число.

```
enum class RequestType {
  ADD,                  //По умолчанию 0
  REMOVE,               //По умолчанию 1
  NEGATE                //По умолчанию 2
};                      // ADD < REMOVE < NEGATE

void ProcessRequest(
    set<int>& numbers,
    RequestType request_type,
    int request_data) {
  if (request_type == RequestType::ADD) {
    numbers.insert(request_data);
  } else if (request_type == RequestType::REMOVE) {
    numbers.erase(request_data);
  } else if (request_type == RequestType::NEGATE) {
    if (numbers.count(request_data) == 1) {
      numbers.erase(request_data);
      numbers.insert(-request_data);
    }
  }
}

ProcessRequest(numbers, RequestType::ADD, 8);
```

```
ProcessRequest(numbers, RequestType::NEGATE, 8);
ProcessRequest(numbers, RequestType::REMOVE, -8);
```

Чем же так хороши enum'ы?

1. Получаем новый тип - тезка enum'а, который можно использовать в качестве параметра функции.
2. Значения типа должны предваряться префиксом с именем (в данном случае `RequestType::`), поэтому вызов `ProcessRequest(numbers, ADD, 8)` не сработает, следовательно `ADD` свободная операция.
3. Значения типа можно сравнивать с помощью `!=`, `==`, а также `<` и `>`. Предпоследнее позволяет использовать `enum'ы` в качестве ключей словарей и элементов множества.
4. `static_cast<RequestType>(0) == RequestType::ADD`.
5. `int32_t x = SomeEnum::VALUE;` - **ошибка компиляции**
6. `enum'ы` очень удобно использовать в `switch`-конструкциях:

```
enum class RequestType {
  ADD,
  REMOVE,
  NEGATE
};

void ProcessRequest(
    set<int>& numbers,
    RequestType request_type,
    int request_data) {
  switch (request_type) {
  case RequestType::ADD:
    numbers.insert(request_data);
    break;
  case RequestType::REMOVE:
    numbers.erase(request_data);
    break;
  case RequestType::NEGATE:
    if (numbers.count(request_data) == 1) {
      numbers.erase(request_data);
      numbers.insert(-request_data);
    }
    break;
  default:
    cout << "Unknown request" << endl;
  }
}
```

Кортежи и пары

Кортежи

Кортеж - структура данных, позволяющая определить несколько типов данных в один. Пример:

```
#include <iostream>
#include <tuple>
```

```
using namespace std;

int main(void) {
  //tuple t = make_tuple(1, "Hello!", 7); В C++17
  tuple<const int, const string, const int> t = make_tuple(1, "Hello!", 7); //
Сработает в кортеже сами элементы
  cout << get<1>(t) << endl;
  return 0;
}
/*Output:
 *Hello!
 */
```

Отличия `tie` и `make_tuple`

- `tie` - создает кортеж из ссылок:

```
auto t = tie(7, "C++", true); // не сработает
```

```
int x = 7;
string s = "C++";
bool b = true;
auto t = tie(x, s, b); //сработает в кортеже ссылки на элементы
```

- `make_tuple` - создает кортеж из самих значений

```
auto t = make_tuple(7, "C++", true); //сработает в кортеже сами элементы
```

```
int x = 7;
string s = "C++";
bool b = true;
auto t = make_tuple(x, s, b); //Сработает в кортеже сами элементы
```

Пары

Кортеж из двух типов с удобным обращениям к элементам:

```
#include <iostream>
#include <tuple>
#include <utility>

using namespace std;

int main(void) {
  pair<int, string> p(1, "Hello!");
  cout << p.first << " " << p.second << endl;
  return 0;
}
/*Output:
 *1 Hello!
 */
```

Альтернативные способы создания

1. `auto p = make_pair(7, "C++")`
2. `pair p(7, "C++")` ⟵ в `C++17`

Давно забытое старое

В белом поясе мы пробовали итерироваться по `map` и там как раз и вылезали пары:

```
#include <iostream>
#include <map>

using namespace std;

int main(void) {
  map<int, string> digits = {{1, "one"}};
  for (const pair<int, string>& e : digits) {
      cout << e.first << " " << e.second << endl;
  }
  return 0;
}
/*Output:
 *1 one
 */
```

Возврат нескольких значений из функции

Пусть некая `foo()` возвращает кортеж, тогда удобно работать с ним так:

```
tuple<bool, string> foo();

int main(void) {
  bool x;
  string s;
  tie(x, s) = foo(); //tie(x, s) имеет тип tuple<bool&, string&> поэтому в x и s запишется то, что нам надо
  cout << x " " << s << endl;
}
```

В `C++17` можно и **удобнее**:

```
tuple<bool, string> foo();

int main(void) {
  auto [x, s] = foo(); //распаковываем кортеж в x и s
  cout << x " " << s << endl;
}
```

Шаблоны функции

Введение в шаблоны

Определим универсальную функцию возведения в квадрат, которая принимает на вход любой тип с определенной операцией умножения самого на себя:

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

template <typename T>
T Sqr(T x) {
  return x * x;
}

int main(void) {
  cout << Sqr(2.5) << endl;  //6.25
  cout << Sqr(2) << endl;    //4
  return 0;
}
```

А теперь определим для пары почти любых элементов шаблонное умножение:

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

template <typename First, typename Second>
pair<First, Second> operator*(const pair<First, Second>& lhs, const pair<First,
Second>& rhs) {
  //Определим переменные объявленных типов First и Second, а затем создадим из
них пару
  First f = lhs.first * rhs.first;
  Second s = lhs.second * rhs.second;
  return make_pair(f, s);
}

template <typename T>
T Sqr(T x) {
  return x * x;
}

int main(void) {
  cout << Sqr(2.5) << endl;  //6.25
  cout << Sqr(2) << endl;    //4
  auto p1 = make_pair(3.5, 4);
  auto res = Sqr(p1);
  cout << res.first << " " << res.second << endl; // 12.25 16
  return 0;
}
```

Универсальные функции вывода контейнеров в поток

```
#include <iostream>
#include <sstream>
#include <vector>
#include <map>
#include <set>
```

```cpp
using namespace std;

template <typename Collection>
string Join(const Collection& c, char d) {
  stringstream ss;
  bool first = true;
  for (const auto& e : c) {
    if (!first) {
      ss << d << e;
    } else {
    first = false;
    ss << e;
    }
  }
  return ss.str();
}

template <typename First, typename Second>
ostream& operator<<(ostream& out, const pair<First, Second>& p) {
  return out << '(' << p.first << "," << p.second << ')';
}

template <typename Key, typename Value>
ostream& operator<<(ostream& out, const map<Key, Value>& m) {
  return out << '{' << Join(m, ',') << '}';
}

template <typename E>
ostream& operator<<(ostream& out, const set<E>& s) {
  return out << '<' << Join(s, ',') << '>';
}

template <typename T>
ostream& operator<<(ostream& out, const vector<T>& v) {
  return out << '[' << Join(v, ',') << ']';
}


int main(void) {
  vector<int> v = {1, 2, 3};
  vector<double> v2 = {1, 2.2, 3};
  vector<vector<int>> v3 = {{1, 2, 3}, {5, 6, 7, 8}};
  map<int, string> m = {{1, "Hello "}, {2, "world! "}};
  cout << v << endl;
  cout << v2 << endl;
  cout << v3 << endl;
  cout << m << endl;
  return 0;
}
/*Output:
 *[1,2,3]
 *[1,2.2,3]
 *[[1,2,3],[5,6,7,8]]
 *{(1,Hello ),(2,world! )}
 */
```

Указание шаблонного параметра-типа

Следующий пример вызовет **ошибку компиляции!**

```
#include <iostream>
using namespace std;
//В КОДЕ ОШИБКА, КОД НЕ КОМПИЛИРУЕТСЯ
template <typename T>
T Max(T a, T b) {
  if (a < b) {
    return b;
  }
  return a;
}

int main(void) {
  cout << Max(2.3, 3) << endl;
  return 0;
}
```

Происходит это потому, что компилятор не понимает, который из типов `double` (2.3) и `int` (3) выбрать в качестве шаблонного параметра `T`. Чтобы это исправить используют следующий приём, который работает и с функцией `max` стандартной библиотеки:

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

template <typename T>
T Max(T a, T b) {
  if (a < b) {
    return b;
  }
  return a;
}

int main(void) {
  cout << Max<double>(2.3, 3) << endl; //Явно указали какой тип выбрать
  return 0;
}
/*Output:
 *3
 */
```

Вторая Неделя Жёлтого Пояса

Простейший способ Юнит-Тестирования

```
#include <iostream>
//Добавляем библиотеку для ассертов
#include <cassert>
using namespace std;

int Sum(int a, int b) {
    return a + b;
}
```

```
void TestSum() {
    //Непосредственное тестирование
    assert(Sum(2, 3) == 5);
    assert(Sum(-2, 3) == 1);
    assert(Sum(2, -3) == -1);
    assert(Sum(-2, -3) == -5);
    assert(Sum(3, -3) == 0);
    assert(Sum(4, -3) == 1);
    assert(Sum(0, -3) == -3);
    assert(Sum(3, 0) == 3);
    cout << "TestSum(): OK" << endl;
}

int main(void) {
    TestSum();
    return 0;
}
/*Output:
 *TestSum(): OK
 */
```

Пусть теперь функция `int Sum(int a, int b)` написана неправильно:

```
int Sum(int a, int b) {
    return a - b;
}
```

Тогда в консоли мы увидим

```
<programm_name_here>: <file_name_here>.cpp:13: void TestSum(): Assertion `Sum(2, 3) == 5' failed.
```

Улучшенный способ Юнит-Тестирования

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

//Два шаблонных типа, чтобы сравнивать, например, int и int_64t
//Контракт шаблона : шаблонный тип можно вывести с помощью <<
template <typename T, typename U>
void AssertEqual (const T& t, const U& u, const string& hint) {
  if (t != u) {
    ostringstream os;
    os << "Assert failed: " << t << " != " << u << " Hint: " << hint << " ";
    throw runtime_error(os.str());
  }
}

//Для сохранения стиля assert(bool b) напишем функцию Assert:
void Assert(bool b, const string& hint) {
  AssertEqual(b, true, hint);
}
```

```
int main(void) {
  try {
    AssertEqual(2, 3, "Int equality");
  } catch (const exception& e) {
    cout << e.what() << endl;
  }
  try {
    Assert(2 == 3, "Int equality 2");
  } catch (const exception& e) {
    cout << e.what() << endl;
  }
  return 0;
}
/*Output
 *Assert failed: 2 != 3 Hint: Int equality
 *Assert failed: 0 != 1 Hint: Int equality 2
 */
```

Теперь, в случае срабатывании **assert'a**, исключение можно перехватить и продолжить выполнение тестов. Строка `hint` подсказывает какой из тестов в программе упал. Можно заметить, что в функции `main` происходит **дублирование кода** с `try - catch` конструкциями, чтобы этого не происходило можно написать новую функцию, которая будет запускать наши тесты:

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

template <typename T, typename U> void AssertEqual (const T& t, const U& u, const
string& hint);
void Assert(bool b, const string& hint);
int Sum (int a, int b);
void Simple_Test();
template <typename TestFunc> void RunTest(TestFunc func, const string&
func_name);

//Два шаблонных типа, чтобы сравнивать, например, int и int_64t
//Контракт шаблона : шаблонный тип можно вывести с помощью <<
template <typename T, typename U>
void AssertEqual (const T& t, const U& u, const string& hint) {
  if (t != u) {
    ostringstream os;
    os << "Assert failed: " << t << " != " << u << " Hint: " << hint << " ";
    throw runtime_error(os.str());
  }
}

//Для сохранения стиля assert(bool b) напишем функцию Assert:
void Assert(bool b, const string& hint) {
  AssertEqual(b, true, hint);
}

int Sum (int a, int b) {
  return a + b - 1;
```

```
}

void Simple_Test() {
  AssertEqual(Sum(2, 3), 5, "Sum of 2 and 3");
  Assert(Sum(3, 0) == 3, "Sum of 3 and 0");
}

template <typename TestFunc>
void RunTest(TestFunc func, const string& func_name) {
  try {
    func();
    //Успешное или неуспешное завершение тестирование будем отправлять в поток
ошибок, чтобы не загромождать stdout
    cerr << func_name << " OK" << endl;
  } catch (const exception& e) {
    cerr << func_name << " fail: " << e.what() << endl;
  }
}

int main(void) {
  RunTest(Simple_Test, "Simple_Test");
  return 0;
}
/*Output (stderr)
 *Simple_Test fail: Assert failed: 4 != 5 Hint: Sum of 2 and 3
 */
```

Однако теперь **при возникновении ошибки тестирования наша программа не завершается**, а продолжает своё выполнение. Если бы после строки в функции `main` был бы остальной код, использующий `int Sum(int a, int b)`, то мы бы получили неверный ответ. Это **плохо** - ошибка в Юнит Тестировании должна быть *звонкой*! Исправим это "завернув" функцию `template <typename TestFunc> void RunTest(TestFunc func, const string& func_name)` в отдельный класс:

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

template <typename T, typename U> void AssertEqual (const T& t, const U& u, const
string& hint);
void Assert(bool b, const string& hint);
int Sum (int a, int b);
void Simple_Test();
void Simple_Test2();
void Simple_Test3();
template <typename TestFunc> void RunTest(TestFunc func, const string&
func_name);
void TestAll();

//Два шаблонных типа, чтобы сравнивать, например, int и int_64t
//Контракт шаблона : шаблонный тип можно вывести с помощью <<
template <typename T, typename U>
void AssertEqual (const T& t, const U& u, const string& hint) {
  if (t != u) {
    ostringstream os;
```

```
    os << "Assert failed: " << t << " != " << u << " Hint: " << hint << " ";
    throw runtime_error(os.str());
  }
}

//Для сохранения стиля assert(bool b) напишем функцию Assert:
void Assert(bool b, const string& hint) {
  AssertEqual(b, true, hint);
}

int Sum (int a, int b) {
  return a + b - 1;
}

//Три простых теста для функции Sum
void Simple_Test() {
  AssertEqual(Sum(2, 3), 5, "Sum of 2 and 3");
  Assert(Sum(3, 0) == 3, "Sum of 3 and 0");
}

void Simple_Test2() {
  AssertEqual(Sum(-2, 3), 1, "Sum of -2 and 3");
  Assert(Sum(-3, 0) == -3, "Sum of -3 and 0");
}

void Simple_Test3() {
  AssertEqual(Sum(0, 0), 0, "Sum of 0 and 0");
  Assert(Sum(-3, 3) == 0, "Sum of -3 and 3");
}


class TestRuner {
private:
  size_t fail_count = 0;

public:
  //Шаблонный метод классаЮ запускающий наши тесты и считающий количество упавших
тестов
  template <typename TestFunc>
  void RunTest(TestFunc func, const string& func_name) {
    try {
      func();
      cerr << func_name << " OK" << endl;
    } catch (const exception& e) {
      cerr << func_name << " fail: " << e.what() << endl;
      fail_count++;
    }
  }

  //Деструктор, который, в случае хотя бы одного упавшего теста, завершает работу
программы.
  ~TestRuner() {
    if (fail_count > 0u) {
      cerr << fail_count << " tests failed. Terminate" << endl;
      exit(1);
    }
  }
};
```

```
//Функция, которая с помощью нового класса запускает все тесты
void TestAll() {
  TestRuner tr;
  tr.RunTest(Simple_Test, "Simple_Test");
  tr.RunTest(Simple_Test2, "Simple_Test2");
  tr.RunTest(Simple_Test3, "Simple_Test3");
  //В данной строчке объект TestRunner tr "разрушается"
}

int main(void) {
  TestAll();
  //Код тут уже не выполнится, если хоть один тест упал
  return 0;
}
/*Output (stderr)
 *Simple_Test fail: Assert failed: 4 != 5 Hint: Sum of 2 and 3
 *Simple_Test2 fail: Assert failed: 0 != 1 Hint: Sum of -2 and 3
 *Simple_Test3 fail: Assert failed: -1 != 0 Hint: Sum of 0 and 0
 *3 tests failed. Terminate
 */
```

Итоги

Как пользоваться?

Минимальный набор:

```
...

void TestSmth() {
  AssertEqual(a, b, hint);
  ...
}

void TestAll() {
  TestRunner tr;
  //Написать сюда все тесты через tr.RunTest(func, func_name);
  tr.RunTest(TestSmth, "TestSmth");
  ...
}

int main(void) {
  TestAll();
}
```

Третья Неделя Жёлтого Пояса

Правило одного определения

Рассмотрим простую ситуацию:

`test.cpp` ⟵ `Header.h` ⟶ `Header.cpp`

Header.h:

```
#pragma once

int Sum(int a, int b);
int Mul(int a, int b);
```

Header.cpp

```
#include "Header.h"

int Sum (int a, int b) {
  return a + b;
}

int Mul (int a, int b) {
  return a * b;
}
```

test.cpp

```
#include "Header.h"
#include <iostream>
using namespace std;

int main(void) {
  cout << "Sum 2 and 3 is: " << Sum(2, 3) << " Mul is: " << Mul(2, 3) << endl;
  return 0;
}

/*Output:
 *Sum 2 and 3 is: 5 Mul is: 6
 */
```

И всё бы ничего, но вот наша прыть молодецкая не даёт нам покоя, и мы - зачем-то - хотим выполнить некий "ре-фак-торинг" нашего кода:

Header.h

```
#pragma once

//Заместо объявления функции написали её определение
int Sum (int a, int b) {
  return a + b;
}

int Mul(int a, int b);
```

Header.cpp

```
#include "Header.h"

//Определение функции Sum тут нет, мы же не хотим нарушать правило одного
определения, верно?

int Mul (int a, int b) {
  return a * b;
}
```

test.cpp

```
#include "Header.h"
#include <iostream>
using namespace std;

int main(void) {
  cout << "Sum 2 and 3 is: " << Sum(2, 3) << " Mul is: " << Mul(2, 3) << endl;
  return 0;
}

/*ERROR!:
... в функции «Sum(int, int)»:
.../Header.h:3: повторное определение «Sum(int, int)»; .../Header.h:3: здесь
первое определение collect2: error: ld returned 1 exit status
*/
```

Получили ошибку, о которой заявил - **Внимание** - ЛИНКОВЩИК! Вот это дела! Как же так получается? Мы же определили `Sum` один раз, а ошибка на повторное определение!

Вспомним архитектуру:

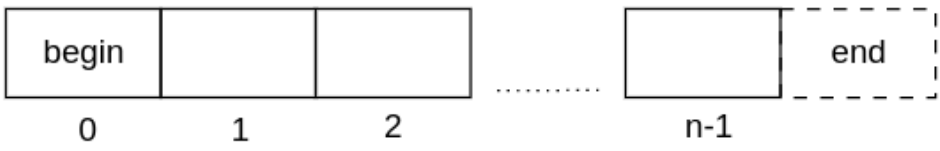
`test.cpp` ⟵ `Header.h` ⟶ `Header.cpp`

Во время изменения кода мы добавили определение `Sum` в `Header.h`, затем препроцессор добавил это определение в два файла `.cpp`: `test.cpp` и `Header.cpp`. Затем наступил этап компиляции кода в двух этих `.cpp` файлах, после чего за дело взялся линковщих, который и обнаружил в двух разных объектных файлах два определения функции `Sum` и выдал нам ошибку!

Четвертая Неделя Жёлтого Пояса

Введение в итераторы

Итератор - способ задать позицию в контейнере. Если у нас есть некий контейнер `c`, то чаще всего к его первому элементу можно обратиться в виде `begin(c)`, также можно обратиться к элементу, который находится **за последним элементом контейнера**, с помощью `end(c)`. Таким образом:



Итератор на примере вектора

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

//begin(vector_name) имеет тип vector<string>::iterator
void PrintVect(vector<string>::iterator range_begin, vector<string>::iterator
range_end) {
  for (vector<string>::iterator it = range_begin;
       it != range_end;
       it++)
  {
    //it++ позволяет перейти к следующему элементу вектора
    //Чтобы получить значение по итератору нужно напечатать звёздочку
    cout << *it << " ";
  }
  cout << "\n";
}

int main() {
  vector<string> v = {"Hi", "My", "Name", "Is", "Slim", "Shady"};
  PrintVect(begin(v), end(v));
  return 0;
}
/*Output:
 *Hi My Name Is Slim Shady
 */
```

Считается, что итераторы более универсальны, чем `range-based for`. Например, с помощью итераторов довольно просто вывести вектор в обратном порядке:

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

int main() {
  vector<string> v = {"Hi", "My", "Name", "Is", "Slim", "Shady"};
  vector<string>::iterator it = end(v);
  while (it != begin(v)) {
    it--;
    cout << *it << ' ';
  }
  cout << "\n";
  return 0;
}
/*Output:
 *Shady Slim Is Name My Hi
 */
```

Тем не менее, `range-based for` использовать предпочтительней, поскольку код будет более читабельным.

Что НЕЛЬЗЯ делать с итераторами?

1. `*end(c)` - undefined behavior, так как `end(c)` это не последний элемент `c`, а нечто после него.
2. `auto it = end(c); ++it;` - undefined behavior, можем залезть не в свою память.
3. `auto it = begin(c); --it;` - undefined behavior, можем залезть не в свою память.