Неделя 4

Потоки. Исключения. Перегрузка операторов

4.1. ООП: Примеры

4.1.1. Практический пример: класс «Дата»

Часто приходится иметь дело с датами. Дата представляет собой три целых числа. Логично написать структуру:

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

Эту структуру можно использовать следующим образом:

```
Date date = {10, 11, 12};
```

Напишем функцию PrintDate, которая принимает дату по константной ссылке (чтобы избежать лишнего копирования):

```
void PrintDate(const Date& date) {
  cout << date.day << "." << date.month << "."
       << date.year << "\n";
}
```

Вывести созданную выше дату можно следующим образом:

```
PrintDate(date);
```

Существует некоторая путаница при таком способе инициализации переменной типа Date. Не понятно, если не видно определения структуры,

какая дата имеется в виду. По такой записи нельзя сразу сказать, где день, месяц и год.

Решить эту проблему можно, создав конструктор. Причем конструктор должен будет принимать не три целых числа в качестве параметров (так как будет точно такая же путаница), а «обертки» над ними: объект типа Day, объект типа Month и объект типа Year.

```
struct Day {
  int value;
};

struct Month {
  int value;
};

struct Year {
  int value;
};
```

Эти типы представляют собой простые структуры с одним полем. Конструктор типа Date имеет вид:

```
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;
  }
};
```

После этого прежняя запись перестает работать:

```
error: could not convert '{10, 11, 12}' from '<brace-enclosed
  initializer list>' to 'Date'
```

Это вынуждает записать такой код:

```
Date date = {Day(10), Month(11), Year(12)};
```

По этому коду мы явно видим, где месяц, день или год. Если перепутать местами месяц и день, компилятор выдаст сообщение об ошибке:

```
could not convert '{Month(11), Day(10), Year(12)}' from '<brace-
  enclosed initializer list>' to 'Date'
```

Однако легко забыть, что все это делалось для лучшей читаемости кода, и «улучшить» код, удалив явные указания типов Day, Month, Year.

```
Date date = {{10}, {11}, {12}};
```

При этом он продолжает компилироваться.

Чтобы сделать код более устойчивым к таким «улучшениям», напишем конструкторы для структур Day, Month, Year.

```
struct Day {
  int value;
  Day(int new_value) {
    value = new_value;
  }
};

struct Month {
  int value;
  Month(int new_value) {
    value = new_value;
  }
};

struct Year {
  int value;
  Year(int new_value) {
    value = new_value;
  }
};
```

Пока что лучше не стало: нежелательный синтаксис все еще можно использовать. Стало даже еще хуже: теперь можно опустить внутренние фигурные скобки (как было в исходном варианте).

```
Date date = {10, 11, 12};
```

Написав такие конструкторы, мы разрешили компилятору неявно преобразовывать целые числа к типам Day, Month, Year. Чтобы избежать неявного преобразования типов, нужно указать компилятору, что так делать не надо, использовав ключевое слово explicit.

```
struct Day {
  int value;
  explicit Day(int new_value) {
    value = new_value;
```

```
  }
};

struct Month {
  int value;
  explicit Month(int new_value) {
    value = new_value;
  }
};

struct Year {
  int value;
  explicit Year(int new_value) {
    value = new_value;
  }
};
```

Ключевое слово explicit не позволяет вызывать конструктор неявно.

4.1.2. Класс Function: Описание проблемы

Допустим, при реализации поиска по изображениям ставится задача упорядочить результаты поисковой выдачи, учитывая качество и свежесть картинок. Таким образом, каждая картинка характеризуется двумя полями:

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

Учет этих двух полей при формировании поисковой выдачи производится с помощью функции ComputeImageWeight и зависит от набора параметров:

```
struct Params {
  double a;
  double b;
};
```

Вес изображения мы определяем следующим образом:

$$weight = quality - freshness * a + b$$

Ниже дан код функции ComputeImageWeight:

```
double ComputeImageWeight(const Params& params,
                          const Image& image) {
  double weight = image.quality;
  weight -= image.freshness * params.a + params.b;
  return weight;
}
```

После этого оказывается, что нужно также учесть рейтинг изображения, то есть каждая картинка характеризуется уже тремя полями:

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

Учет рейтинга при формировании веса изображения контролируется с помощью нового параметра:

```
struct Params {
  double a;
  double b;
  double c;
};
```

И производится также в функции ComputeImageWeight:

```
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;
}
```

Если вдруг кроме функции ComputeImageWeight существует функция ComputeQualityByWeight:

```
double ComputeQualityByWeight(const Params& params,
                              const Image& image,
                              double weight) {
  double quality = weight;
  quality += image.freshness * params.a + params.b;
  return quality;
}
```

то нужно не забыть внести изменения и в нее тоже:

```
double ComputeQualityByWeight(const Params& params,
                              const Image& image,
                              double weight) {
  double quality = weight;
  quality -= image.rating * params.c;
  quality += image.freshness * params.a + params.b;
  return quality;
}
```

В данном случае присутствует неявное дублирование кода: существует некоторый способ вычисления веса изображения от его качества. Он представлен в коде два раза: в функции ComputeImageWeight и в функции ComputeQualityByWeight.

4.1.3. Класс Function: Описание

Чтобы убрать дублирование, нужно для сущности «способ вычисления веса изображения от его качества» написать некоторый класс. По сути, эта сущность есть некоторая функция, поэтому реализовывать нужно класс Function и функцию MakeWeightFunction, которая будет возвращать нужную функцию.

```
Function MakeWeightFunction(const Params& params,
                            const Image& image) {
...
}
```

Функции ComputeImageWeight и ComputeQualityByWeight следует переписать следующим образом:

```
double ComputeImageWeight(const Params& params,
                          const Image& image) {
  Function function = MakeWeightFunction(params, image);
  return function.Apply(image.quality);
}

double ComputeQualityByWeight(const Params& params,
                              const Image& image,
                              double weight) {
  Function function = MakeWeightFunction(params, image);
  function.Invert();
  return function.Apply(weight);
}
```

При этом в функции ComputeQualityByWeight нужно использовать обратную функцию.

Функция MakeWeightFunction, таким образом, должна быть реализована следующим образом:

```
Function MakeWeightFunction(const Params& params,
                            const Image& image) {
  Function function;
  function.AddPart('-',
                   image.freshness * params.a + params.b);
  function.AddPart('+', image.rating * params.c);
  return function;
}
```

Метод AddPart добавляет часть функции к объекту типа Function.

4.1.4. Класс Function: Реализация

Реализуем класс Function. Этот класс обладает методами:

Метод AddPart — добавляет очередную часть в функцию. Принимает два аргумента: символ операции и вещественное число.

Метод Apply — возвращает вещественное число, применяя текущую функцию к некоторому числу. Это константный метод, так как он не должен менять функцию.

Метод Invert — заменяет текущую функцию на обратную.

Приватное поле parts — набор элементарных операций. Каждая элементарная операция представляет собой объект типа FunctionPart.

В качестве конструктора используется конструктор по умолчанию.

В классе FunctionPart понадобится:

Конструктор — принимает на вход символ операции и операнд (вещественное число). Сохраняет эти значения в приватных полях.

Метод Apply — применяет операцию к некоторому числу. Константный метод.

Метод Invert — инвертирует элементарную операцию.

Теперь можно привести реализации обоих классов:

```
class FunctionPart {
public:
  FunctionPart(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:
  void AddPart(char operation, double value){
    parts.push_back({operation, value});
  }
  double Apply(double value) const {
    for (const FunctionPart& part : parts) {
      value = part.Apply(value);
    }
    return value;
  }
  void Invert() {
    for (FunctionPart& part : parts) {
      part.Invert();
    }
    reverse(begin(parts), end(parts));
  }
```

```
private:
  vector<FunctionPart> parts;
};
```

4.1.5. Класс Function: Использование

Проверим, как работают созданные классы. Создадим изображение (объект класса Image) и проинициализируем его поля:

```
Image image = {10, 2, 6};
```

Вычисление веса изображения невозможно без задания параметров формулы. Создадим объект типа Params:

```
Params params = {4, 2, 6};
```

Подсчитаем вес изображения с помощью функции ComputeImageWeight:

```
// 10 - 2 * 4 - 2 + 6 * 6 = 36
cout << ComputeImageWeight(params, image) << endl;
```

В результате получим:

```
36
```

Теперь протестируем функцию ComputeQualityByWeight:

```
// 20 - 2 * 4 - 2 + 6 * 6 = 46
cout << ComputeQualityByWeight(params, image, 46) << endl;
```

На выходе имеем, чему должно быть равно качество изображения:

```
20
```

Таким образом, код работает успешно.

4.2. Работа с текстовыми файлами

4.2.1. Потоки в языке C++

Стандартная библиотека обеспечивает гибкий и эффективный метод обработки целочисленного, вещественного, а также символьного ввода через консоль, файлы или другие потоки. А также позволяет гибко расширять способы ввода для типов, определенных пользователем.

Существуют следующие базовые классы:

istream поток ввода (cin)

ostream поток вывода (cout)

iostream поток ввода/вывода

Все остальные классы, о которых пойдет речь далее, от них наследуются.

Классы, которые работают с файловыми потоками:

ifstream для чтения (наследник istream)

ofstream для записи (наследник ostream)

fstream для чтения и записи (наследник iostream)

4.2.2. Чтение из потока построчно

Чтение из потока производится с помощью оператора ввода (≫) или функции getline, которая позволяет читать данные из потока построчно.

Пусть заранее создан файл со следующим содержимым:

```
hello world!
second line
```

Для работы с файлами нужно подключить библиотеку fstream:

```
#include <fstream>
```

Чтобы считать содержимое файла следует объявить экземпляр класса ifstream:

```
ifstream input("hello.txt");
```

В качестве аргумента конструктора указывается путь до желаемого файла.

Далее можно создать строковую переменную, в которую будет записан результат чтения из файла:

```
string line;
```

Функция getline первым аргументом принимает поток, из которого нужно прочитать данные, а вторым — переменную, в которую их надо записать. Чтобы проверить, что все работает, можно вывести переменную line на экран:

```
getline(input, line);
cout << line << endl;
```

Чтобы считать и вторую строчку, можно попробовать запустить следующий код:

```
getline(input, line);
cout << line << endl;

getline(input, line);
cout << line << endl;
```

Если вызвать getline в третий раз, то она не изменит переменную line, так как уже достигнут конец файла и из него ничего не может быть прочитано:

```
getline(input, line);
cout << line << endl;

getline(input, line);
cout << line << endl;

getline(input, line);
cout << line << endl;
```

Чтобы избежать таких ошибок, следует помнить, что getline возвращает ссылку на поток, из которого берет данные. Поток можно привести к типу bool, причем false будет в случае, когда с потоком уже можно дальше не работать.

Переписать код так, чтобы он выводил все строчки из файла и ничего лишнего, можно так:

```
ifstream input("hello.txt");
string line;
while (getline(input, line)) {
    cout << line << endl;
}
```

Следует обратить внимание, что переводы строки при выводе добавлены искусственно. Это связано с тем, что функция getline, на самом деле, считывает данные до некоторого разделителя, причем по умолчанию до символа перевода строки, который в считанную строку не попадает.

4.2.3. Обработка случая, когда указанного файла не существует

Рассмотрим ситуацию, когда по некоторым причинам неверно указано имя файла или файла с таким именем не может существовать в файловой системе. Например, внесем опечатку:

```
ifstream input("helol.txt");
```

При запуске этого кода оказывается, что он работает, ничего не выводит, но никак не сигнализирует о наличии ошибки.

Вообще говоря, желательно, чтобы программа не умалчивала об этом, а явно сообщала, что файла не существует и из него нельзя прочитать данные.

У файловых потоков существует метод is_open, который возвращает true, если файловый поток открыт и готов работать. Программу, таким образом, следует переписать так:

```
ifstream input("helol.txt");
string line;

if (input.is_open()){
  while (getline(input, line)) {
    cout << line << endl;
  }
  cout << "done!" << endl;
} else {
  cout << "error!" << endl;
}
```

Следует также отметить, что файловые потоки можно приводить к типу bool, причем значение true соответствует тому, что с потоком можно работать в данный момент. Другими словами, код можно переписать в следующем виде:

```
ifstream input("helol.txt");
string line;

if (input){
  while (getline(input, line)) {
    cout << line << endl;
  }
  cout << "done!" << endl;
} else {
  cout << "error!" << endl;
}
```

4.2.4. Чтение из потока до разделителя

Научимся считывать данные с помощью getline поблочно с некоторым разделителем. Например, в качестве разделителя может выступать символ «минус». Допустим, считать нужно дату из следующего текстового файла date.txt:

```
2017-01-25
```

Для этого создадим:

```
ifstream input("date.txt");
```

Объявим строковые переменные year, month, day.

```
string year;
string month;
string day;
```

Нужно считать файл таким образом, чтобы соответствующие части файла попали в нужную переменную. Воспользуемся функцией getline и укажем разделитель:

```
if (input) {
  getline(input, year,  '-');
  getline(input, month, '-');
  getline(input, day,   '-');
}
```

Чтобы проверить, что все работает, выведем переменную на экран через пробел:

```
cout << year << ' ' << month << ' ' << day << endl;
```

4.2.5. Оператор чтения из потока

Решим ту же самую задачу с помощью оператора чтения из потока (≫). Записывать считанные данные будем в переменные типа int.

```
ifstream input("date.txt");
int year = 0;
int month = 0;
int day = 0;
if (input) {
  input >> year;
  input.ignore(1);
  input >> month;
```

```
  input.ignore(1);
  input >> day;
  input.ignore(1);
}
cout << year << ' ' << month << ' ' << day << endl;
```

После того, как из потока будет считан год, следующим символом будет «минус», от которого нужно избавиться. Это можно сделать с помощью метода ignore, который принимает целое число — сколько символов нужно пропустить. Аналогично считываются месяц и день. Получается такой же результат.

То, каким методом пользоваться, зависит от ситуации. Иногда бывает удобнее сперва считать всю строку целиком.

4.2.6. Оператор записи в поток. Дозапись в файл.

Данные в файл можно записывать с помощью класса ofstream:

```
const string path = "output.txt";

ofstream output(path);
output << "hello" << endl;
```

После проверим, что записалось в файл, открыв его и прочитав содержимое:

```
ifstream input(path);
if (input) {
  string line;
  while (getline(input, line)) {
    cout << line << endl;
  }
}
```

Чтобы избежать дублирования кода, имеет смысл создать переменную path.

Для удобства можно создать функцию, которая будет считывать весь файл:

```
void ReadAll(const string& path) {
  ifstream input(path);
  if (input) {
    string line;
    while (getline(input, line)) {
      cout << line << endl;
```

```
    }
  }
}
```

Предыдущаяя программа примет вид:

```
const string path = "output.txt";

ofstream output(path);
output << "hello" << endl;

ReadAll(path);
```

Следует отметить, что при каждом запустке программы файл записывается заново, то есть его содержимое удалялось и запись начиналась заново.

Для того, чтобы открыть файл в режиме дозаписи, нужно передать специальный флажок ios::app (от англ. append):

```
ofstream output(path, ios::app);
output << " world!" << endl;
```

4.2.7. Форматирование вывода. Файловые манипуляторы.

Допустим, нужно в определенном формате вывести данные. Это могут быть имена колонок и значения в этих колонках.

Сохраним в векторе names имена колонок и после этого создадим вектор значений:

```
vector<string> names = {"a", "b", "c"};
vector<double> values = {5, 0.01, 0.000005};
```

Выведем их на экран:

```
for (const auto& n : names) {
  cout << n << ' ';
}
cout << endl;
for (const auto& v : values) {
  cout << v << ' ';
}
cout << endl;
```

При этом читать значения очень неудобно.

Для того, чтобы решить такую задачу, в языке C++ есть файловые манипуляторы, которые работают с потоком и изменяют его поведение. Для того, чтобы с ними работать, нужно подключить библиотеку iomanip.

fixed Указывает, что числа далее нужно выводить на экран с фиксированной точностью.

```
cout << fixed;
```

setprecision Задает количество знаков после запятой.

```
cout << fixed << setprecision(2);
```

setw (set width) Указывает ширину поля, которое резервируется для вывода переменной.

```
cout << fixed << setprecision(2);
cout << setw(10);
```

Этот манипулятор нужно использовать каждый раз при выводе значения, так как он сбрасывается после вывода следующего значения:

```
for (const auto& n : names) {
  cout << setw(10) << n << ' ';
}
cout << endl;
cout << fixed << setprecision(2);
for (const auto& v : values) {
  cout << setw(10) << v << ' ';
}
```

Здесь колонки были выведены в таком же формате.

setfill Указывает, каким символом заполнять расстояние между колонками.

```
cout << setfill('.');
```

left Выравнивание по левому краю поля.

```
cout << left;
```

Для удобства напишем функцию, которая будет на вход принимать вектора имен и значений, и выводить их в определенном формате:

```
void Print(const vector<string>& names,
           const vector<double>& values, int width) {
  for (const auto& n : names) {
   cout << setw(width) << n << ' ';
  }
  cout << endl;
  cout << fixed << setprecision(2);
  for (const auto& v : values) {
   cout << setw(width) << v << ' ';
  }
  cout << endl;
}
```

Покажем как пользоваться манипуляторами setfill и left:

```
cout << setfill('.');
cout << left;
Print(names, values, 10);
```

3.4. Перегрузка операторов для пользовательских типов

В данном видео будет рассмотрено, как сделать работу с пользовательскими структурами и классами более удобной и похожей на работу со стандартными типами. Например, когда целое число считывается из консоли или выводится в консоль, это можно сделать очень удобно — с помощью операторов ввода и вывода.

3.4.1. Тип Duration (Интервал)

Рассмотрим структуру Интервал, которая включает поля: час и минута.

```
struct Duration {
  int hour;
  int min;
}
```

Напишем функцию, которая будет возвращать интервал, считывая значения из потока:

```
Duration ReadDuration(istream& stream) {
  int h = 0;
  int m = 0;
  stream >> h;
  stream.ignore(1);
  stream >> m;
  return Duration {h, m};
}
```

Также определим функцию PrintDuration, которая будет выводить интервал в поток.

```
void PrintDuration(ostream& stream, const Duration&
 ↪  duration) {
  stream << setfill('0');
  stream << setw(2) << duration.hour << ':'
         << setw(2) << duration.min;
}
```

Воспользуемся функциями, сперва заведя и инициализируя строковый поток:

```
stringstream dur_ss("01:40");
Duration dur1 = ReadDuration(dur_ss);
PrintDuration(cout, dur1);
```

Использовать функции ReadDuration и PrintDuration, в принципе, удобно, но было бы удобнее использовать операторы ввода из потока и вывода в поток.

3.4.2. Перегрузка оператора вывода в поток

Определим оператор вывода в поток, который принимает в качестве первого аргумента поток, а в качестве второго константную ссылку на экземпляр объекта. Пусть (пока) он возвращает void:

```
void operator<<(ostream& stream, const Duration& duration) {
  stream << setfill('0');
  stream << setw(2) << duration.hour << ':'
         << setw(2) << duration.min;
}
```

Заметим, что сигнатуры функции PrintDuration и оператора вывода очень похожи, поэтому реализацию можно скопировать без каких-либо изменений.

Мы сделали класс гораздо удобнее для работы:

```
cout << dur1;
```

Но если мы попытаемся добавить перенос на новую строку, программа не скомпилируется.

```
cout << dur1 << endl;
```

Попытаемся понять, почему так происходит. Рассмотрим, например, следующий код:

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

Оператор вывода `operator<<` первым аргументом принимает поток, а вторым — строку для вывода, и возвращает поток, в который делал вывод.

Можно вызвать по цепочке два оператора вывода:

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

Поэтому оператор вывода должен возвращать не void, а ссылку на поток:

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

После этого код начинает работать.

3.4.3. Перегрузка оператора ввода из потока

Аналогичным образом определим оператор ввода из потока:

```
istream& operator>>(istream& stream, Duration& duration)) {
  stream >> duration.h;
  stream.ignore(1);
  stream >> duration.m;
  return stream;
}
```

Теперь считывать интервалы можно прямо из потока:

```
stringstream dur_ss("02:50");
Duration dur1 {0, 0};
dur_ss >> dur1;
cout << dur1 << endl;
```

Оператор ввода также возвращает ссылку на поток, чтобы была возможность считывать сразу несколько переменных.

3.4.4. Конструктор по умолчанию

Дополнительно стоит отметить, что язык C++ позволяет задать значения структуры по умолчанию. Этого можно добиться с помощью создания конструктора по умолчанию:

```
struct Duration {
  int hour;
  int min;

  Duration(int h = 0, int m = 0) {
    hour = h;
    min = m;
  }
}
```

3.4.5. Перегрузка арифметических операций

Реализуем возможность складывать интервалы естественным образом:

```
Duration dur1 = {2, 50};
Duration dur2 = {0, 5};
cout << dur1 + dur2 << endl;
```

Такой код еще не компилируется, поскольку не определен оператор плюс.

Оператор плюс на вход принимает два объекта и возвращает их сумму:

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

Сокращения lhs и rhs обозначают Left/Right Hand Side.

После этого код начинает работать.

```
Duration dur1 = {2, 50};
Duration dur2 = {0, 5};
cout << dur1 + dur2 << endl;
// OUTPUT: 02:55
```

Однако, запустим этот код для другой пары интервалов:

```
Duration dur1 = {2, 50};
Duration dur2 = {0, 35};
cout << dur1 + dur2 << endl;
// OUTPUT: 02:85
```

Интервал «02:85» — достаточно странный интервал. Следует сделать так, чтобы минуты всегда были от 0 до 59. Логично исправить для этого конструктор типа Duration:

```
struct Duration {
  int hour;
  int min;

  Duration(int h = 0, int m = 0) {
    int total = h * 60 + m;
    hour = total / 60;
    min = total % 60;
  }
}
```

После такого определения конструктора:

```
Duration dur1 = {2, 50};
Duration dur2 = {0, 35};
cout << dur1 + dur2 << endl;
// OUTPUT: 03:25
```

3.4.6. Сортировка. Перегрузка операторов сравнения.

Допустим, необходимо для вектора интервалов

```
Duration dur1 = {2, 50};
Duration dur2 = {0, 35};
Duration dur3 = dur1 + dur2;
vector<Duration> v {
  dur1,  dur2,  dur3
}
```

расположить элементы этого вектора по возрастанию.

Для удобства напишем функцию PrintVector:

```
void PrintVector(const vector<Duration>& durs) {
  for (const auto& d : durs) {
    cout << d << ' ';
  }
  cout << endl;
}
```

Использовать эту функцию можно следующим образом:

```
vector<Duration> v {
  dur1,  dur2,  dur3
}
PrintVector(v); // => 03:25 02:50 00:35
```

Попробуем отсортировать вектор:

```
sort(begin(v), end(v));
PrintVector(v);
```

При компиляции возникают ошибки, который говорят о том, что оператор сравнения не определен. Можно исправить эту ошибку двумя способами:

- Определить функцию-компаратор и передать ее в качестве третьего аргумента в функцию sort.

```
bool CompareDurations(const Duration& lhs, const
 ↪  Duration& rhs) {
  if (lhs.hour == rhs.hour) {
    return lhs.min < rhs.min;
  }
  return lhs.hour < rhs.hour
}
```

Пример использования функции компаратора:

```
Duration dur1 { 1, 12 };
Duration dur2 { 1, 13 };
cout << boolalpha << CompareDurations(dur1, dur2) <<
 ↪  endl;
// OUTPUT: true
```

- Перегрузить оператор «меньше» для типа Duration. Если третий аргумент функции sort не указан, при сортировке используется он.

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

С помощью манипулятора потока boolalpha можно выводить в консоль значения логических переменных как true/false.

3.4.7. Использование перегруженных операторов в собственных структурах

Решим для примера практическую задачу. Пусть дан текстовый файл с результатами забега нескольких бегунов:

```
0:32 Bob
0:15 Mary
0:32 Jim
```

Необходимо создать файл, где бегуны будут отсортированы согласно их результату, а также дополнительно вывести бегунов, которые бежали дольше всех.

Для решения этой задачи удобно использовать структуру типа map, поскольку в этом случае автоматически поддерживается упорядоченность данных:

```
ifstream input("runner.txt");
Duration worst;
map<Duration, string> all;
if (input) {
  Duration dur;
  string name;
  while (input >> dur >> name) {
```

```
      if (worst < dur) {
        worst = dur;
      }
      all[dur] += (name + " ");
    }
  }
  ofstream out("result.txt");
  for (const auto durationNames& : all) {
    out << durationNames.first << '\t' << durationNames.second
     ↪  << endl;
  }
  cout << "Worst runner: " << all[worst] << endl;
  // OUTPUT: "Worst runner: Bob Jim"
```

Результирующий файл:

```
0:15  Mary
0:32  Bob Jim
```

3.3. Исключения в C++ (введение)

Исключение — это нестандартная ситуация, то есть когда код ожидает определенную среду и инварианты, которые не соблюдаются.

Банальный пример: функции, которая суммирует две матрицы, переданы матрицы разных размерностей. В таком случае возникает исключительная ситуация и можно «бросить» исключение.

3.3.1. Практический пример: парсинг даты в заданном формате

Допустим необходимо парсить даты

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

из входного потока.

Функция ParseDate будет возвращать объект типа Date, принимая на вход строку:

```
Date ParseDate(const string& s){
  stringstream stream(s);
  Date date;
  stream >> date.year;
  stream.ignore(1);
  stream >> date.month;
  stream.ignore(1);
  stream >> date.day;
  stream.ignore(1);
  return date;
}
```

В этой функции объявляется строковый поток, создается переменная типа Date, в которую из строкового потока считывается вся необходимая информация.

Проверим работоспособность этой функции:

```
string date_str = "2017/01/25";
Date date = ParseDate(date_str)
cout << setw(2) << setfill('0') << date.day << '.'
     << setw(2) << setfill('0') << date.month << '.'
     << date.year << endl; // OUTPUT: "25.01.2017"
```

Код работает. Но давайте защитимся от ситуации, когда данные на вход приходят не в том формате, который ожидается:

```
2017a01b25
```

Программа выводит ту же дату на экран. В таких случаях желательно, чтобы функция явно сообщала о неправильном формате входных данных. Сейчас функция это не делает.

От этой ситуации можно защититься, изменив возвращаемое значение на bool, передавая Date в качестве параметра для его изменения, и добавляя внутри функции нужные проверки. В случае ошибки функция возвращает false. Такое решение задачи очень неудобное и существенно усложняет код.

3.3.2. Выброс исключений в C++

В C++ есть специальный механизм для таких ситуаций, который называется исключения. Что такое исключения, можно понять на следующем примере:

```
Date ParseDate(const string& s){
  stringstream stream(s);
  Date date;
  stream >> date.year;
  if (stream.peek() != '/') {
    throw exception();
  }
  stream.ignore(1);
  stream >> date.month;
  if (stream.peek() != '/') {
    throw exception();
  }
  stream.ignore(1);
  stream >> date.day;
  return date;
}
```

Если формат даты правильный, такой код отработает без ошибок:

```
string date_str = "2017/01/25";
Date date = ParseDate(date_str);
cout << setw(2) << setfill('0') << date.day << '.'
     << setw(2) << setfill('0') << date.month << '.'
     << date.year << endl;
```

Если сделать строчку невалидной, программа упадет:

```
string date_str = "2017a01b25";
Date date = ParseDate(date_str);
cout << setw(2) << setfill('0') << date.day << '.'
     << setw(2) << setfill('0') << date.month << '.'
     << date.year << endl;
```

Чтобы избежать дублирования кода, создадим функцию, которая проверяет следующий символ и кидает исключение, если это необходимо, а затем пропускает его:

```
void EnsureNextSymbolAndSkip(stringstream& stream) {
  if (stream.peek() != '/') {
    throw exception();
  }
  stream.ignore(1);
}
```

Функция ParseDate примет вид:

```
Date ParseDate(const string& s){
  stringstream stream(s);
  Date date;
  stream >> date.year;
  EnsureNextSymbolAndSkip(stream);
  stream >> date.month;
  EnsureNextSymbolAndSkip(stream);
  stream >> date.day;
  return date;
}
```

3.3.3. Обработка исключений. Блок try/catch

Ситуация когда программа падает во время работы не очень желательна, поэтому нужно правильно обрабатывать все исключения. Для обработки ошибок в C++ существует специальный синтаксис:

```
try {
  /* ...код, который потенциально
     может дать исключение... */
} catch (exception&) {
  /* Обработчик исключения. */
}
```

Проверим это на практике:

```
string date_str = "2017a01b25";
try {
  Date date = ParseDate(date_str);
  cout << setw(2) << setfill('0') << date.day << '.'
       << setw(2) << setfill('0') << date.month << '.'
       << date.year << endl;
} catch (exception& ex) {
  cout << "exception happens";
}
```

Хорошо бы донести до вызывающего кода, что произошло и где произошла ошибка. Например, если отсутствует какой-то файл, указать, что файл не найден и путь к файлу. Для этого есть класс runtime_error:

```
void EnsureNextSymbolAndSkip(stringstream& stream) {
  if (stream.peek() != '/') {
    stringstream ss;
    ss << "expected / , but has: " << char(stream.peek());
    throw runtime_error(ss.str());
  }
  stream.ignore(1);
}
```

Если у исключения есть текст, его можно получить с помощью метода what исключения.

```
} catch (exception& ex) {
  cout << "exception happens: " << ex.what();
}
```