## 4.3. 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.

### 4.3.1. 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.

### 4.3.2. 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.

### 4.3.3. 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.

### 4.3.4. 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;
  }
}
```

### 4.3.5. 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 operator+(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
```

### 4.3.6. 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.

### 4.3.7. 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
```