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, Mouth, Year.

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

При этом он продолжает компилироваться.

Чтобы сделать код более устойчивым к таким «улучшениям», напишем конструкторы для структур Day, Mouth, 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, Mouth, 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;
}
```