C++

Браун Екатерина, Захаренко Артем

19 декабря 2020 г.

1 Классы и структуры!!!

1. Оператор присваивания

Давайте разберем следующую конструкцию:

```
#include <iostream>

int main() {
    int a, b, c = 10;
    std::cout << (a = b = c) << std::endl;
    std::cout << a << " " << b << " " << c << std::endl;
    return 0;
}
```

Чтобы понимать, что этот код делает, оператор присваивания нужно воспринимать как бинарный оператор, у которого (как и у других операторов, например, $+, *, >>$ и т.д.) есть возвращаемое значение. В данном случае, начиная с 17 стандарта гарантируется, что код парсится именно так, и порядок вычислений именно такой (до 17 стандарта это UB)

```
#include <iostream>

int main() {
    int a, b, c = 10;
    std::cout << (a = (b = c)) << std::endl;
    std::cout << a << " " << b << " " << c << std::endl;
    return 0;
}
```

То есть сначала вычислится выражение (b = c), в b запишется c, потом вернется значение, которое лежит в b. Это значение запишется аналогичным образом в a и опертаор =, вызванный для a вернет значение, которое лежит в a. В итоге на первой строке выведется одна десятка, на следующей 3 десятки. Подробнее, почему нужно ставить скобки в std::cout и про приоритет операторов тут:

Разберем несколько примеров, как можно самостоятельно перегрузить оператор = (перегружать его можно только внутри класса) :

```
struct Foo {
  int x;
  Foo (int x_ = 0): x(x_) {}
  void operator=(const Foo &other) {
    x = other.x;
  }
};
```

```
struct Foo {
  int x;
  Foo (int x_ = 0): x(x_) {}
  Foo operator=(const Foo &other) {
    x = other.x;
    return *this;
  }
};
```

```
struct Foo {
  int x;
  Foo (int x_ = 0): x(x_) {}
  Foo& operator=(const Foo &other) {
    x = other.x;
    return *this;
  }
};
```

(a) Вроде бы обычный оператор присваивания, делает что нужно (присваивает), и самые тривиальные конструкции с ним будут работать. Но подобный пример для Foo, который был в начале, уже словит ошибку компиляции.

(b) Теперь мы возвращаем значение, которое лежит в объекте, который мы только что перезаписали. Уже лучше, код из самого начала (аналогичный для Foo) будет работать.

(c) Вернули по ссылке. Теперь не будет лишного копирования. Может показаться, что в данном случае плохо возвращать значение по ссылке, если у нас элемент временный, но на самом деле временный объект живет до ";"и все будет хорошо. То есть с последним вариантом работают следующие конструкции:

```cpp
#include <iostream>
struct Foo {
  int x;
  Foo (int x_ = 0): x(x_) {}
  Foo& operator=(const Foo &other) {
    x = other.x;
    return *this;
  }
};

int main() {
  Foo a, b, c;
  a = b = c = Foo(5);
  (a = b) = (c = Foo(6));
  std::cout << a.x << std::endl; // 6
  std::cout << b.x << std::endl; // 5
  std::cout << c.x << std::endl; // 6
}
```

Такая конструкция тоже работает, но зачем так делать????

```cpp
struct Foo {
  int x;
  Foo (int x_ = 0): x(x_) {}
  int operator=(const Foo &other) {
    x = other.x;
    return 228;
  }
};
```

К слову, оператор "+="тоже бинарный оператор, который что-то возвращает, и аналогичные конструкции с ним тоже работают.

2. Немного про макросы

```cpp
assert(vec == std::vector<int>{1, 2})
assert((vec == std::vector<int>{1, 2}))
```

Первая строка не скомпилируется, так как assert это макрос, а макросы это штука древняя, про то, что нужно балансировать фигурные скобки внутри аргументов макросы не в курсе. Вторая строка вполне себе работает, так как про баланс круглых скобок макросам известно (привет ПСП).

3. Обратно к классам, aggregate initialization

Пусть у нас есть достаточно примитивный класс. То есть в нем нет конструктора, все поля публичные, тогда есть такая вот фича:

```cpp
#include <iostream>
#include <vector>
#include <cassert>

struct Foo {
  int x = 10;
  int y = 20;
  char z = 'a';
  std::vector <int> vec = {1, 2};
};

int main() {
  Foo a;

  assert(a.x == 10);
  assert(a.y == 20);
  assert(a.z == 'a');
  assert((a.vec == std::vector<int>{1, 2}));

  Foo b{100, 200};
  assert(b.x == 100);
  assert(b.y == 200);
  assert(b.z == 'a');
  assert((b.vec == std::vector<int>{1, 2}));

  Foo c{1, 2, 'b', {3, 4, 5}};
  assert(c.x == 1);
  assert(c.y == 2);
  assert(c.z == 'b');
  assert((c.vec == std::vector<int>{3, 4, 5}));
}
```

Важно то, как мы инициализировали переменные типа Foo. Никакого конструктора у нас нет, мы пользовались aggregate initialization, то есть в фигурных скобочках прописали первые сколько-то значений. Именно столько значений проинициализировались в переменной, причем важен порядок (в том, в котором объявлены внутри структуры, в таком и инициализируются). Остальные поля останутся как были, с дефолтными значениями. Работает эта фича c++ по-разному в стандартах 03, 11, 14, 17, 20 :)

Таким же образом можно проинициализировать массив:

```cpp
int a[10] = {1};
//1 0 0 0 0 0 0 0 0 0
```

4. Про константные поля внутри классов

Рассмотрим следующую структуру:

```
struct Foo {
   const int x;
};
```

Сначала кажется что все хорошо, пока мы с этой структурой не начинаем работать. Из интересных фактов, теперь мы не можем ее копировать. В дефолтном операторе присваивания все поля просто копируются, то есть нам надо будет переприсвоить константное поле, чего делать нельзя (в этом и весь смысл констант). Если очень надо, то можете сами ручками переопределить оператор присваивания.

Для констант работает правило, что их нельзя переприсваивать, но можно один раз проинициализировать. Для примера, первый код вполне себе работает, а вот второй уже нет:

```
struct Foo {
    const int x = 10;
    Foo(int x_) : x(x_) {
    }
};
```

```
struct Foo {
    const int x = 10;
    Foo(int x_) : {
      x = x_;
    }
};
```

Во втором случае мы пытаемся переприсвоить константу, в первом просто сразу ее инициализируем. aggregate inintialization для данного случая тоже работает. Будем считать, что инициализировать переменную можно не более одного раза. Что очень важно для констант, они обязаны быть проинициализированы.

5. Порядок внутри member initializer list

И еще 2 примера рабочего кода, и кода, который UB (первый - рабочий, второй - UB)

```
struct Foo {
    int x, y;
    Foo(int a) : x(a), y(x) {}
};
```

```
1 struct Foo {
2     int x, y;
3     Foo(int a): y(a), x(y) {}
4 };
```

Порядок при инициализации не такой, в котором вы инициализируете внутрик конструктора (чего, вероятно, хотелось бы ожидать во втором случае), а тот, в котором поля объявленны внутри стурктуры. Поэтому в обоих случаях сначала инициализируется x, потом y. Из-за этого во втором примере в x сначала запишется непроинициализированное значение y, потом в y запишем a. В итоге получили UB.