Новые спецификаторы в C++11, копия, оригинал здесь [http://archive.kalnitsky.org/2012/01/23/introduction-to-cpp11-new-specifiers/]

Ну вот и закончились новогодние праздники. Жизнь постепенно возвращается в привычное русло и вместе с тем я продолжаю писать в блог.

Сегодня я опять продолжу тему нового стандарта C++ и расскажу о некоторых нововведениях, которые будут весьма полезны разработчикам классов. Речь пойдет о спецификаторах, предоставленных C++11. А именно: `override`, `final`, `default` и `delete`.

Спецификатор override

Понятие переопределения, думаю, известно всем. Здесь я не буду описывать *что это* и *как проявляется*. Я лишь отмечу об одной проблеме (или, скорее, особенности) старого стандарта касательно переопределения функций в классе наследнике.

Рассмотрим следующую ситуацию.

```
1. class Base
2. {
3. public:
4.     virtual void doSomething(int x);
5. };
6. // ...
7. class Derived : public Base
8. {
9. public:
10.     virtual void doSomething(long x);
11.  };
```

Что мы имеем? У нас есть некоторый базовый класс и класс-наследник. Допустим, пользователь захотел изменить в классе-наследнике поведение метода `doSomething()`. Он его переопределяет, но по некой причине (невнимательности, например) нечаянно указал другой тип аргумента: `long` вместо `int`. Компилятор на это не ругнется, но код не будет работать так, как это запланировал автор класса.

Все дело в том, что методы обладают различными сигнатурами и, в данном случае, произойдет перекрытие методов. Перекрытие — это отдельная тема. Сейчас я лишь отмечу, что работая через указатель/ссылку на базовый класс, будет вызываться метод определенный в базовом классе, но никак не метод, переопределенный нами.

Необходимо будет потратить определенное время, чтобы отыскать ошибку. Новый же стандарт C++ вводит ключевое слово **override**, которое позволяет отслеживать подобного рода ошибки и переводить их на ошибки времени компиляции.

```
1. class Base
2. {
3. public:
4.     virtual void doSomething(int x);
5. };
6. // ...
7. class Derived : public Base
```

```
8. {
9. public:
10.      virtual void doSomething(long x) override;
11.  };
```

Иными словами, компилятор, обнаружив `override`, проверяет существование метода с данной сигнатурой в базовом классе. Если же такого метода нет — выдает ошибку.

Спецификатор final

C++11 позволяет запрещать в классах-наследниках переопределение определенных методов. Достигается это за счет применения спецификатора **final** рядом с сигнатурой метода.

```
1. class Base
2. {
3. public:
4.     virtual void doSomething(int x) final;
5. };
6. // ...
7. class Derived : public Base
8. {
9. public:
10.      virtual void doSomething(int x); // ошибка!
11.  };
```

Данный спецификатор также позволяет запрещать наследование от некоторого класса.

```
1. class Base final {};
2. class Derived : public Base {}; // ошибка!
```

Спецификатор `final` издавна существует в Java. Наконец он появился и в C++.

Спецификатор default

Полезность данного спецификатора весьма спорная: кто-то найдет его полезным, а кому-то он покажется бесполезным. Так или иначе, суть его заключается в том, что пользователь может указать компилятору реализовать ту или иную функцию-член класса по-умолчанию. Что имеется ввиду? Предположим есть класс:

```
1. class Foo
2. {
3. public:
4.     Foo(int x) {/* ... */}
5. };
```

Как видно, класс имеет один пользовательский конструктор, а значит конструктор по-умолчанию сгенерирован не будет. Дабы стала возможна запись вида:

```
Foo Obj;
```

пользователю необходимо определить конструктор без параметров.

```
1. class Foo
2. {
3. public:
```

```
4.      Foo() {}
5.      Foo(int x) {/* ... */}
6. };
```

Вместо определения конструктора без параметров, в C++11 появилась возможность просто указать компилятору сгенерировать его по-умолчанию. Достигается это, как я сказал выше, при помощи спецификатора `default`.

```
1. class Foo
2. {
3. public:
4.      Foo() = default;
5.      Foo(int x) {/* ... */}
6. };
```

Реализация по-умолчанию более эффективна, чем реализация определенная пользователем. Но при нынешних системах, я не думаю что затраты на пользовательский конструктор буду заметны. В любом случае, об этом спецификаторе стоит знать.

Стоит отметить, что он применим только к *специальным* функциям-членам. К специальным относятся:

- конструктор по-умолчанию;
- конструктор копий;
- конструктор перемещения (введен в C++11);
- оператор присваивания;
- оператор перемещения (введен в C++11);
- деструктор.

Спецификатор delete

Данный спецификатор более полезный, нежели спецификатор `default`. Он призван пометить те методы, работать с которыми нельзя. То есть, если программа ссылается явно или неявно на эту функцию — ошибка на этапе компиляции. Запрещается даже создавать указатели на такие функции.

```
1. class Foo
2. {
3. public:
4.      void baz() = delete;
5. };
```

С помощью этого спецификатора, можно легко запретить конструктор копий (который уже все привыкли прятать в `private`) или запретить автоматическое приведение типов.

```
1. class Foo
2. {
3. public:
4.      Foo() = default;
5.      Foo(const Foo&) = delete;
6.      void bar(int) = delete;
7.      void bar(double) {}
8. };
9. // ...
10.  Foo obj;
```

```
11.  obj.bar(5);      // ошибка!
12.  obj.bar(5.42);   // ok
```

Можно также запретить оператор `new`:

```
1. class Foo
2. {
3. public:
4.     void *operator new(std::size_t) = delete;
5.     void *operator new[](std::size_t) = delete;
6. };
7. // ...
8. Foo* ptr = new Foo; // ошибка!
```

Заключение

К сожалению, все рассмотренные спецификаторы кроме `default` и `delete` поддерживаются начиная с *g++-4.7*, который мне так и не удалось найти под Ubuntu. Поэтому `override` и `final` тестировались на компиляторе *clang++ 2.9*.