Конструирование в C++11. Часть 1: Стирая границы

16. ноября 2012 разработка

Очень долго процесс конструирования объектов оставался неизменным, что-то было унаследовано от старого доброго C, что-то было добавлено благодаря появлению классов. Шли годы и в старых методах конструирования были выявлены серьезные изъяны. Большая часть которых, несомненно, является чисто синтаксической(т.е. наличие изъяна ни каким образом на производительность не влияла). В настоящей статье речь пойдёт о новшествах, появившихся в новом стандарте C++, относящихся к новому синтаксису инициализации. Остальные новшества будут рассмотрены в части 2.

Единообразная инициализация

Экскурс в историю

Все мы знаем, что для инициализации статического массива некоторым набором элементов существуют крайне простой и удобный синтаксис:

```
1  int a[10] = {1, 2, 3, 4, 5, 6,};
```

Вышеописанный массив инициализирует первые 6 элементов значениями, которые описаны в фигурных скобках, оставшиеся же элементы заполняются нулями(т.е. значениями по умолчанию).

Кроме массивов, подобным способом инициализации могли похвастаться и различные *простые(POD)* структуры данных:

```
1  struct A
2  {
3      int a;
4      int b;
5  };
6
7  class B
8  {
9  public:
10     int a;
11     int b;
12 };
13
14 union C
15 {
16     int a;
17     int b;
18 };
19
20 int main()
21 {
22     A a = {1, 2};
23     B b = {2, 3};
24     C c = {1};
25 }
```

```
22
23
24
25
```

И на этом на список можно завершить, т.к. всё удобство инициализации было строго ограничено рамками вышеозначенных способов. Никакого вмешательства в процесс инициализации C++, образца 2003 года, не терпел. Очень многие, в то время, задавались вопросом: ”доколе пользовательские типы данных будут считаться второсортными?”. Наверно, многим из вас приходилось писать код, в тестах или еще где, очень похожий на следующий:

```
1  std::vector<int> vec;
2  vec.push_back(1);
3  vec.push_back(7);
4  vec.push_back(-3);
5  vec.push_back(4);
6  vec.push_back(-15);
7  vec.push_back(22);
8  ...
```

Несомненно, был выход писать это по другому:

```
1  int a[] = {1, 7, -3, 4, -15, 22 ...};
2  std::vector<int> vec(&a[0], &a[sizeof(A)/sizeof(int) - 1]);
```

Но, согласитесь, писать 2 строчки вместо одной удовольствие не из приятных. К тому же это добавляло сущностей, которые к делу отношения не имеют совершенно.

Еще одним важным моментом, здесь, является факт, что использование фигурных скобок всегда давало жестко регламентированный результат(о котором написано выше), в то время как для некой динамической инициализации всегда использовались пользовательские конструкторы и, соответственно, круглые**()** скобки. В связи с этой особенностью связан любопытный баг, который очень часто вводил(вводит?) в ступор новичков в C++, да и, откровенно говоря, не только новичков. Давайте рассмотрим следующий код:

```
1  class A
2  {
3  public:
4      A(): m_i(7)
5      {}
6  private:
7      int m_i;
8  };
9
10 int main()
11 {
12     A a();//#1
13 }
```

Что делает строчка **#1**? Новичок уверенно отвечает: “Определение объекта **a** типа **A**, который конструируется посредством вызова конструктора по умолчанию!”. В логике ему не окажешь, и такой ответ должен бы быть правильный, но, к сожаление таковым не является. Правильным ответом будет: “Объявление функции **a** которая не принимает аргументов и возвращает объект типа **A**”. Вот такие вот весёлые последствия использования круглых скобок в различных контекстах.

Новое время

Понимая всю плачевность ситуации с проблемами описанными выше, комитет стандартизации разродился унифицированным синтаксисом инициализации. И в борьбе круглых и фигурных скобок победили последние. Начиная разматывать клубок с последней проблемы, продемонстрируем как она решается в новом стандарте:

```
class A
{
public:
    A(): m_i(7)
    {  }
private:
    int m_i;
};

int main()
{
    A a{};
    A b = {};
};
```

Вуаля! Заменяем круглые скобки на фигурные и наша запись больше никак не может трактоваться двояко. Это определение объекта с вызовом конструктора по умолчанию! Разумеется, данная нотация справедлива не только для конструктора по умолчанию, но и для любого другого:

```
class A
{
public:
    A(int i, int j, int k): m_i(i), m_j(j), m_k(k)
    {  }
private:
    int m_i;
    int m_j;
    int m_k;
};

int main()
{
    A a{1, 2, 3};//Эквивалентно A a(1, 2, 3)
};
```

Кроме того, фигурные скобки могут быть использованы без имени типа, к примеру в операторе *return*:

```
A foo()
{
    return {1, 2, 3};//Эквивалентно return A(1, 2, 3)
}

```

Или как параметр функции:

```
void foo(A a)
{
    a;
}

int main()
{
    foo({1, 2, 3});//Эквивалентно foo(A(1, 2, 3))
};


```

Неправда-ли удобная нотация? Теперь нет нужды постоянно повторять тип, компилятор уже его знает(прям как с *auto*)! Просто конструируем то, что нам нужно с использование фигурных скобок. Хотя в примерах я использую константные значения, вместо них, с таким же успехом могут быть переменные и объекты.

Кстати, старое, доброе выделение памяти под массив теперь тоже можно совмещать с инициализацией:

```
int* a = new int[5]{1, 2, 3, 4, 5};
```

При всём удобстве новой нотации, не обошлось и без “капли дёгтя в бочке мёда”. Одним из недостатков подобной нотации является запрет на*“сужение”(narrowing)* типа. Означает это следующие: Если тип аргумента в фигурных скобках отличается от типа, который ожидает конструктор и, следовательно, требуется неявное преобразование, то следующее преобразование запрещено:

1. Число с плавающей точкой в целое
2. *Из long double* в *double*, или *float* и из *double* во *float*, за исключением случаев, когда это константное значение и оно умещается в “суженный” тип.
3. Из целого числа(или *enum*) в число с плавающей запятой, за исключением случав, когда это константное значение и оно умещается в тип с плавающей точкой.
4. *Из* целого числа(или *enum*) более высокого порядка, в целое число более низкого(например из *long long* в *long*), за исключением случаев, когда это константное значение и оно умещается в “суженный” тип.

Примеры:

```
int i1 = {1};//верно
int i2 = {1L};//верно
int i3 = {9999999999999999999L};//ошибка, п.4
long long longI{0};//верно
int i4 = {longI};//ошибка п.4
int i5{1.0};//ошибка п.1
int i6{1.0f};//ошибка п.1
float f1{1.0};//верно
double doubleF{1.0};//верно
float f2{doubleF};//ошибка п.2
float f3{9999999999999999999.0};//ошибка п.2
float f4{9999999999999999999L};//ошибка п.3
float f5{99999L};//верно
float f6{longI};//ошибка п.3
```

Хотя для кого-то это далеко не дёготь(для меня к примеру), а торжество “буквы закона”. Ведь подобное поведение всегда сопровождалось предупреждениями в любом уважаемом компиляторе. И тот факт, что с новым синтаксисом предупреждение превратилось в ошибку лишь добавляет уверенности в корректности кода(на тот случай, если по каким-либо причинам проект, с которым вы работаете не использует флаг, который интерпретирует все предупреждения как ошибки). Хотя данная особенность и не является дёгтем для всех, есть и другая особенность, но проявляющаяся в некотором другом случае, который мы рассмотрим чуть позже.

Рассмотрев новый синтаксис мы, попутно, решили последнюю проблему из нашего списка(списка из двух проблем :)), но, всё же, осталось непонятным как прикрутить этот новый синтаксис к первой проблеме, проблеме заполнения вектора и ему подобных. Не будешь же писать конструктор на бесконечное количество аргументов, пусть даже с существующими *variadic templates*. Комитет по стандартизации, тоже решил, что пора облегчить жизнь страждущим и в результате появился*std::initializer_list<T>*

Список инициализации

Начнем с того, что новый синтаксис для заполнения массивов типа*std::vector*, и даже *std::map* всё таки появился. Никаких отличий от того, чем раньше обладали лишь статические массивы нет:

```
std::vector<int> vec = {1, 2, 3, 4 ,5 , 6};
std::map<std::string, int> map = {{"first", 1}, {"second", 2}, {"t
```

Никаких *push_back*’ов и *insert*’ов! Правда не забывайте о “сужении”, здесь оно точно так же действует. Никаких отличий. И в этом вся сила нового, унифицированного синтаксиса. Теперь множество различных(но смежных) действий производится с использованием одного синтаксиса.

Теперь давайте разберемся как такое стало возможным, т.е. как *std;::vector*может принимать подобный список и наполняться, за счёт него, элементами. Это стало возможным за счёт нового, специально введенного типа *std::initialzer_list<T>*, который находится в заголовке *<initializer_list>*.Дабы обозначить, что наш тип(в нашем случае *vector*) может принимать списки неопределенного размера, мы должны

определить специальный конструктор, который принимает *std::initialzer_list* в качестве единственного аргумента, или же все последующие аргументы имеют значения по умолчанию. Т.е. новый конструктор для вектора будет выглядеть так:

```
vector(const std::initializer_list<T>& list)
{
    ....
}

```

или так:

```
vector(std::initializer_list<T> list)
{
    ....
}

```

для *std::map* конструктор будет выглядеть так:

```
map(std::initializer_list<pair<T, U>> list)
{
    ....
}

```

initializer_list является довольно легковесным типом и вы сами можете в этом убедится посмотрев его заголовок. Вся основная “магия” нового синтаксиса сокрыта в недрах компилятора и нам предоставляется тонкий интерфейс к оной. Происходит же это примерно так: создается некий промежуточный массив, в который помещаются все элементы из списка инициализации(**{...}**). После этого конструируется *initializer_list*, который хранит в себе указатель на начало и конец этого временного массива. После чего *initializer_list* передается в конструктор и мы можем делать с ним всё, что захотим.

Давайте рассмотрим простенький пример со списком:

```
#include <iostream>
#include <initializer_list>

class A
{
public:
    A(std::initializer_list<int> list)
    {
        for(auto& item : list)
        {
            std::cout << "item=" << item << "\n";
        }
    }
};

int main()
{
    A a{23,321,321,3,213,213,12};
```

```
};
```

Т.к. *initializer_list* имеет методы *begin* и *end*, которые возвращают необходимые итераторы мы можем использовать его в цикле *for* нового формата. Также мы можем иметь несколько конструкторов с *initializer_list*, разных типов:

```
class A
{
public:
    A(std::initializer_list<int> list)
    {
        for(auto& item : list)
        {
            std::cout << "Integral item=" << item << "\n";
        }
    }
    A(std::initializer_list<std::string> list, int def = 0)
    {
        for(auto& item : list)
        {
            std::cout << "String item=" << item << "\n";
        }
    }
};

int main()
{
    //A(std::initializer_list<int> list)
    A a{23,321,321,3,213,213,12};
    //A(std::initializer_list<std::string> list, int def = 0)
    A b{"one", "two", "three", "eleven"};
};
```

Есть в стандарте и особое правило, которое регламентирует то, как конструктор с *initializer_list* участвует в перегрузке: если используется синтаксис списка инициализации(**{...}**), то конструктор с *initializer_list* имеет превосходство перед любыми другими конструкторами, если список не пуст. Если список пуст, то вызывается конструктор по умолчанию, если он присутствует. И вот тут мы находим наст��ящий кусок дёгтя.

Рассмотрим пример:

```
std::vector<int> vec{5};
//#1
assert(vec.size() == 5);
assert(vec[0] == 0);
//#2
```

```
assert(vec.size() == 1);
assert(vec[0] == 5);
```

Как вы полагаете, какой набор *assert*'ов сработает? Разумеется все готовы к подвоху и отвечают **#1**. Вы абсолютно правы, а происходит это потому, что конструкторы с *initializer_list* имеют превосходство, а значит {5} это не “создать вектор из 5 элементов”, а “создать вектор из одного элемента и присвоить этому эдементу 5”. Поэтому для создания вектора из 5-и элементов придется использовать старый синтаксис с круглыми скобками. Более того, придется всегда следить, что у вас нет перегруженных конструкторов, которые могут быть “случайно заменены”, конструктором со списком. Или же, если таковые есть, использовать тип очень аккуратно и помнить о том, чем это может быть чревато. Довольно серьёзный недостаток, но с ним, к сожалению, ничего не поделаешь. В остальном же новый синтаксис просто шикарен.

Резюме

Итак, в современном C++ сущности могут быть инициализированы одним из следующих методов:

1. Старый способ, использующий фигурные скобки.
2. Инициализация списком, т.е. вызов конструктора с *initializer_list*
3. Прямое конструирование, т.е. вызов конструктора без *initializer_list* у класса, или же инициализация значением у базового типа.
4. Старый способ, использующий круглые скобки.

```
PodStruct pod = {1, 2, 3};//#1
InitStruct initialized = {1, 2, 3};//#2
int array[] = {1, 2, 3};//#1
int multiArray[2][2] = {{1, 2}, {3, 4}};//#1
int* pArray = new int[3]{1, 2, 3};//#3
InitStruct default{};//#3
InitStruct nonDefault{1.0};//#3
int i = {};//#3
int j{5};//#3
std::map<int, int> map = {{1, 2}, {3, 4}, {5, 6}};//#2
```

Конструирование в C++11. Часть 2: Работа над ошибками

18. декабря 2012 разработка

В части 1 мы рассмотрели новый синтаксис, а теперь пришло время к тому, что я называю “работой над ошибками”. Всё представленное в данной статье так или иначе, как мне кажется, является доработкой концепций, которые были просто-напросто просмотрены в первом стандарте.

Равноправие членов

Каждый из нас знает простой синтаксис инициализации константных членов класса:

```
class A
{
    const int a = 42;
};
```

При этом, любое неконстантное поле, которое требуется инициализировать, необходимо инициализировать в конструкторе:

```
class A
{
    const int a = 42;
    int references;
public:
    A(): references(0)
    {

    }
};
```

Странная несправедливость, вы не находите? На мой взгляд это простая недоработка изначального стандарта, которая, наконец, исправлена в новом стандарте C++11.

Теперь права константных и не-константных членов класса уравнены и они могут быть инициализированы в определении класса:

```
class A
{
    const int a = 42;
    int references = 0;
};
```

Заметьте, что конструктор нам больше не нужен, т.к. единственным его назначением была инициализация поля **references,** которое теперь успешно инициализировано без необходимости определения собственного конструктора; мы можем положиться на генерируемый конструктор по умолчанию. Довольно

удобная конструкция, особенно когда у нас много конструкторов, где надо инициализировать одно и то же поле. Давайте рассмотрим несколько вариаций на тему инициализации в определении класса:

```
struct A
{
    int b = 1;
    int c{b + 1};
};

int main()
{
    A obj;
    std::cout << obj.b << " " << obj.c;
};

```

Несложно догадаться, что этот код выведет

```
1 2
```

А что если изменить порядок?

```
struct A
{
    int c{b + 1};
    int b = 1;
};
```

Что выведет этот код? Правильно,- неизвестно что, т.к. члены класса всегда конструируются в том порядке, в котором их декларировали. С появлением такого способа инициализации новичкам, как мне кажется, будет проще запомнить это правило: *Всегда инициализируй члены класса в том порядке, в котором они объявлены.* Ведь здесь действует тоже правило, что и для списка инициализации членов в конструкторе. Более того, новый способ инициализации, по сути, лишь “синтаксический сахар”, надстройка над списком инициализации членов в конструкторе, т.е. компилятор вместо программиста “подставляет” код инициализации в каждый конструктор. Вышеприведенный код абсолютно эквивалентен нижеприведенному:

Версия выводящая

```
1 2
```

```
struct A
{
    int b;
    int c;

    A(): b(1), c(b + 1),
    {
    }
```

```
};


```

И выводящая непредсказуемый результат:

```
struct A
{
    int c;
    int b;

    A(): b(1), c(b + 1),
    {
    }
};

```

Кстати, вы заметили, что при инициализации **c** мы использовали **b + 1**, т.е. мы использовали *выражение*, а, следовательно, мы можем использовать любой код при инициализации, так же как мы использовали в списке инициализации. К примеру. мы можем инициализировать **b**результатом возвращенным из функции:

```
int answer()
{
    return 42;
}

struct A
{
    int b = answer();
};

```

Или, если чей-то извращенный ум пожелает, то можно написать и так:

```
int answer()
{
    return 42;
}

struct A
{
    int b = answer();
    int c{[this](){return b + 1;}()};
};

```

В продолжение, рассмотрения нового способа инициализации членов, немного усложним пример:

```
struct A
{
```

```
    int b = 1;
    int c{b + 1};
    int d;
    int a = d++;
    A(int value): d(value)
    {
    }
};

int main()
{
    A obj{3};
    std::cout << obj.a << " " << obj.b << " " << obj.c << " " <<
};
```

Тут всё тоже очевидно, этот код выведет:

```
3 1 2 4
```

А что выведет код, если мы немного его изменим:

```
struct A
{
    int b = 1;
    int c{b + 1};
    int d;
    int a = d++;
    A(int value): d(value), a(5)
    {
    }
};
```

может быть он выведет

```
5 1 2 4
```

?

Нет, он выведет

```
5 1 2 3
```

!

И в этом поведении скрывается одно очень важно свойство, которое можно описать следующим образом:

1. Инициализация в списке членов инициализации в конструкторе имеет более высокий приоритет по

отношению к инициализации в определении класса.

2. Если присутствует инициализация в списке членов инициализации в конструкторе, то инициализация в определении класса попросту игнорируется!

Т.е. мало того, что само значение из инициализации в определении класса будет проигнорировано, так еще и никакие побочные эффекты, которые могли бы быть вызваны этой инициализацией не будет произведены! Именно поэтому выражение **d++** никогда не было выполнено и мы имеем тройку в хвосте нашего вывода. Помните об этом свойстве, оно достаточно коварно.

Выражайтесь яснее

Еще одно новшество стандарта связано с пользовательским операторами преобразования типов:

```
struct A
{
    operator int()
    {
        return 42;
    }
};
```

Все в этой конструкции замечательно и покрывает все нужды по преобразованию типов, да вот беда, неявное преобразование типов, которое становится возможным благодаря наличию подобного оператора. Давайте рассмотрим безобидный пример:

```
struct A
{
    operator int()
    {
        return 42;
    }

    operator char*()
    {
        return "A";
    }
};

int main()
{

    std::cout << A() << "\n";
};
```

В MSVS этот код выводит **42,** а это явно не то, что нам хотелось. Оставим за бортом рассуждения как лучше реализовывать получения имени класса и остановимся на нашем примере. Конечно, мы можем использовать явное преобразование при выводе имени, но, лично для меня, это слишком тяжкий груз.

Получается чересчур много ненужной писанины. Гораздо проще этот пример можно решить с помощью нового стандарта:

```
struct A
{
    explicit operator int()
    {
        return 42;
    }

    operator char*()
    {
        return "A";
    }
}
```

Обратите внимание на ключевое слово *explicit* перед оператором, именно оно даёт нам нужное поведение. Теперь, при добавлении любого нового оператора преобразования мы помечаем его *explicit* и оставляем только один оператор, который может быть использован неявно – **operator char*()**. И наша проблема решена.

Значение ключевого слова *explicit* точно такое же как и в случае с конструктором,- если желаете преобразовать тип A в тип B извольте указать это явно. Вот и всё.

Кому-то может показаться, что данное нововведение не существенно, но я бы не стал считать его таковым. К примеру, все мы знаем, что получить C-строку из *std::string* можно с помощь. *c_str()*, но было бы неплохо иметь оператор преобразования в C-строку. Понятно почему его нет до сих пор – если бы он был, то он бы возвращал “нестабильный” указатель в местах, где этого никто не ждёт. Но сейчас, когда пользователь может явно указать, что он хочет, не разрешая никаких неявных преобразований, возможно, он всё таки появится.

Еще одним примером является библиотека Qt, которая содержит следующие макросы:

QT_NO_CAST_FROM_ASCII

QT_NO_CAST_TO_ASCII

QT_NO_CAST_FROM_BYTEARRAY

Все они относятся к преобразованию типов и могут быть заменены на более элегантное решение с помощь.нового, явного преобразования типов.

Царь-конструктор

Я думаю многие(если не каждый) сталкивался с необходимостью писать функцию *init* для класса, в котором имеется более одного конструктора. Более того, я полагаю, многие из вас сталкивались с тем, что забывали вызывать эту функцию *init* в одном из конструкторов и это было чревато самыми неприятными последствиями. Все эти потуги с *init* выглядели настолько чуждо элегантной структуре классов, что если бы

новый стандарт не решил данную проблему, раз и навсегда, это вызывало бы недоумение, мягко говоря.

> Хотя возможность инициализации не константных членов класса частично решает данную проблему, остаются варианты, когда необходимо таки получить какие-то аргументы через конструктор. Да и очень часто инициализация выносится из заголовка специально, дабы не раздувать зависимости. Поэтому не будем считать данную возможность решением проблемы функции init

Решением в данной ситуации послужило уже давно обкатанное на других языках средство – делегирующий конструктор. Суть его проста: любой конструктор может вызвать любой другой перегруженный конструктор данного класса отличный от самого себя. Последнее требование вполне логично, ведь если конструктор будет сам себя вызывать мы получим рекурсивный вызов конструктора, что есть нонсенс. Так что данное поведение заслужило свою строку в стандарте – оно запрещено.

Рассмотрим пример:

```
class Rectangle
{
public:
    Rectangle(size_t width, size_t height):
        m_Width(width),
        m_Height(height)
    {
        std::cout << "Target ctor\n";
    }
    Rectangle(size_t width):
        Rectangle(width, width)
    {
        std::cout << "Delegate ctor\n";
    }
private:
    size_t m_Width;
    size_t m_Height;
};

int main()
{
    Rectangle square(10);
};
```

Как вы понимаете, этот код даст следующий вывод:

```
Target ctor
Delegate ctor
```

Ни убавить, ни прибавить - всё достаточно просто. Цепочка конструкторов может быть любой длины, при этом основным конструктором считается первый вызванный конструктор. Из этого вытекает одно очень

важное свойство. Все мы знаем, что если конструктор не был завершен за счёт брошенного исключения, то и деструктор не будет вызван. Здесь ничего не меняется. Но, в связи с появлением цепочки конструкторов, появилась новая ситуация, когда основной конструктор отработал, а, в последствии, в цепочке было брошено исключение. Так вот, если основной конструктор отработал, то деструктор будет вызван, в не зависимости от завершённости остальных конструкторов в цепочке.

```
class Rectangle
{
public:
    Rectangle(size_t width, size_t height):
        m_Width(width),
        m_Height(height)
    {
        std::cout << "Rectangle(size_t width, size_t height)\n";
    }
    Rectangle(size_t width):
        Rectangle(width, width)
    {
        std::cout << "Rectangle(size_t width)\n";
        throw std::string();
    }
    Rectangle(std::string):
        Rectangle(42)
    {
        std::cout << "Rectangle(std::string)\n";
    }
    ~Rectangle()
    {
        std::cout << "~Rectangle()\n";
    }
private:
    size_t m_Width;
    size_t m_Height;

};

int main()
{
    try
    {
        Rectangle square("SQUARE");
    }
    catch(...)
    {

    };
};
```

Таким образом, за счёт делегирования конструкторов можно навсегда избавиться от

написания *init* функций. Единственный случай, теперь, когда она может понадобиться это когда необходим вызов виртуальной функции при инициализации, т.к. эта проблема никуда, к сожалению, не ушла.

Конструкторонаследование

Еще одной ситуацией, которая заставляет многих C++ программистов писать “лишний” код является наследование, а именно: наследование конструкторов. Очень часто в классе наследнике определяется конструктор, только для того, чтобы вызывать конструктор базового класса. Отличным примером тут является иерархия QObject в Qt. Так каждый класса потомок, который хочет иметь возможность участвовать в местном авто-удалении должен передать родительский объект в консруктор класса QObject, который выглядит так:

```
QObject(QObject* pParent = nullptr)
```

Т.е. вы можете переопределить конструктор и иметь функционал авто-удаления или не переопределять и не иметь оного. Есть и другие случаи, когда у вас нет выбора и вы обязаны переопределять конструктор, т.к. необходимо получить некоторые параметры. Да что говорить, ситуации, при которой приходится определять такие конструкторы-заглушки только для удовлетворения потребностей родителя повсеместны. И это отнюдь не добавляет радости программистам C++.

Раз уж мы заговорили об этой проблеме, то, естественно, новый стандарт решает и её. И вот как выглядит это решение:

```
using A::A;
```

Где **A** это имя класса, конструкторы которого мы наследуем. Пример:

```
class A
{
public:
    explicit A(size_t i): m_i(i)
    {
        std::cout << "A(size_t i)\n";
    }
private:
    size_t m_i;
};

class B: public A
{
public:
    using A::A;
};

int main()
{
    B b(55);
};
```

```
20
21
```

Теперь давайте разберём, что же происходит при использовании данной*using* директивы: Происходит неявное объявление всех конструкторов класса родителя в классе потомке, за исключением копирующего и перемещающего конструкторов. Они не наследуются. Более того, существует ряд случаев, когда количество унаследованных конструкторов будет больше, чем родитель имеет оных. Все эти случаи связаны с параметрами по умолчанию и эллипсами(…). Так, если родитель имеет некий конструктор с параметрами по умолчанию, то на основании этого конструктора будет сгенерирован и унаследован набор конструкторов, в котором, помимо оригинала, будут присутствовать конструкторы с “обрезанным” количеством параметров. Где обрезанными могут быть параметры по умолчанию и эллипсы. Пример:

```
1  class A
2  {
3  public:
4      explicit A(int i, int j = 12, int k = 35, ...)
5      {
6      }
7  private:
8      size_t m_i;
9  };
10
11
12 class B: public A
13 {
14 public:
15     using A::A;
16 };
```

Класс **B** унаследует следующий набор конструкторов:

- explicit B(int i, int j = 12, int k = 35, …)
- explicit B(int i, int j = 12, int k = 35)
- explicit B(int i, int j = 12)
- explicit B(int i)

И, в довесок, будет иметь набор собственных конструкторов:

- B()
- B(B&&)
- B(const B&)

Заметьте, что

1. *explicit* наследуется вместе с конструктором; это, также. верно для*constexpr*, *delete* и спецификации исключений.
2. Конструктор по умолчанию генерируется неявно, даже с учётом наличия унаследованного конструктора.

Еще одно свойство, которое стоит упомянуть, заключается в том, что если пользователь объявит один из конструкторов родителя явно, то неявной генерации произведено не будет.

Кроме того, всё вышесказанное относится и к конструкторам-шаблонам; они наследуются по тем же правилам.

Равняйсь!

Завершить статью мне бы хотелось упоминанием о появлении двух новых операторов связанных с выравниванием:

- alignof – позволяет узнать по какой границе выравнивается тот или иной тип или объект
- alignas – позволяет задать границу по которой должен быть выравнен тип или объект.

Оператор *alignas* имеет одно ограничение: он не подействует если при помощи него осуществляется попытка выставить выравнивание, которое является более слабым(т.е. с меньшим значением), чем подразумевает тип или объект, к которому этот оператор применяется. Также, если применяется несколько *alignas*, то выигрывает более сильный. Например:

```
class alignas(4) A
{
    long long variable;
};

int main()
{
    std::cout << alignof(A) << "\n";
    alignas(16) alignas(32) A a;
    std::cout << alignof(A);
};
```

Будет выведено 8(на x86 архитектуре) и 32.

Оператор *alignas* может принимать в качестве операнда как неотрицательное целое, являющееся степенью 2-ки, так и имя типа. Если операндом выступает тип, то это эквивалентно следующей записи:

```
alignas(alignof(T))
```

Вместе с этими операторами появился целый ворох основанных на них компонентов STL, которые я не вижу смысла рассматривать отдельно:*std::alignment_of, std::aligned_storage, std::aligned_union, std::align* и так далее.