Почему пара и кортеж (С++) — это чаще всего плохо

Многим программистам знакомы концепции пар и кортежей (pair и tuple) — их реализации есть в STL, Boost (и может быть где-нибудь еще). Для тех, кто не знает, что это такое, я коротко поясню — это шаблоны, позволяющие сгруппировать несколько значений (пара — только 2, tuple — много) с целью хранить\передавать\принимать их вместе.   
Пример из MSDN:

pair <int, double> p1 ( 10, 1.1e-2 );

pair <int, double> p2 = make\_pair ( 10, 2.22e-1 );

cout << "The pair p1 is: ( " << p1.first << ", " << p1.second << " )." << endl;

cout << "The pair p2 is: ( " << p2.first << ", " << p2.second << " )." << endl;

Поначалу идея кажется заманчивой, ведь:

1. Вместо передачи в функцию нескольких векторов одинаковой размерости можно передать только один вектор пар\кортежей, не заботясь о проверке их соответствия.
2. Можно легко вернуть из функции набор значений, не мороча голову с указателями или ссылками в out-параметрах (для многих это сложно)
3. Можно избежать создания кучи мелких структур из 2-3 полей (меньше кода — лучше).

Но есть и тёмная сторона этой силы.  
  
Как говориться в известной поговорке — простота хуже воровства. Пары и кортежи — как раз тот случай. Они действительно дают все описанные выше преимущества. Но давайте задумаемся какой ценой. 

**Содержимое пары или кортежа — загадка**

Давайте посмотрим на объявление вот такой функции:

pair<string, string> GetInfo( int personId );

Что бы Вы думали она возвращает? Имя и фамилию? А с чего Вы взяли? Может быть — номер паспорта и код в налоговой? А может быть полное имя и номер телефона. Или реальное имя и никнейм. Мысль в том, что нигде в объявлении функции не описывается, что будет содержаться в возвращаемой паре. Вы, конечно, помните это сейчас. А вспомните через год? Кроме того, другому программисту, который решит воспользоваться этой функцией, придётся лезть в её код и выискивать, что же она возвращает — а уж это и вовсе ставит крест на всём ООП-подходе, модульности, инкапсуляции и прочих важных вещах.  
Сравните вышеуказанный код со следующим:

struct Person

{

string Name;

string Surname;

}

Person GetInfo( int personId );

Всё кристально ясно, читать код функции для понимания возвращаемого значения нет нужды. 

**Порядок данных в паре или кортеже — загадка**

Ок, мы хорошо подумали над прошлым примером и переписали нашу функцию вот так:

pair<string, string> GetNameAndSurname( int personId );

Теперь чётко ясно, что возвращает она имя и фамилию человека. Но вот в чём вопрос — в каком порядке? Вам кажется вполне очевидным какой-то определённый порядок этих строк в паре, но у меня для Вас плохая новость. Вы живёте в мире, где люди пользуются разным порядком слов в именах, разными форматами дат и времён, пишут как слева направо, так и наоборот, ездят по дорогам с разносторонним движением и т.д. Тот факт, что Вам кажется единственно возможным только этот вариант порядка значений в паре не доказывает ровным счётом ничего. Как говорит один из законов Мёрфи — "**Если что-нибудь может быть истолковано несколькими способами, оно будет истолковано именно самым неверным из них**".   
В случае использования отдельной структуры (класса) для возвращаемого значения мы всегда имеем однозначную трактовку кода.

**Плохая расширяемость**

Идём дальше — что будет, если со временем в нашу функцию мы захотим добавить еще данных? Да, мы можем заменить пару на кортеж и наращивать его до абсудрного размера:

tuple<string, string, int> GetNameAndSurnameAndBirthday( int personId );

Но какой же это кошмар для всех, кто этой функцией пользуется! Получается, что после каждого её изменения нужно пересматривать все вызовы функции, проверяя, к правильным ли полям мы обращаемся. Ужас.

**Невозможность показать отсутствие одного из значений**

Иногда в наборе значений одно или несколько полей могут быть не установленными. Это очень легко отобразить в структуре или классе (завести переменную isSet или написать метод проверки поля), но совершенно невозможно отобразить в паре или кортеже, где предполагается, что набор содержит все значения и они валидны. В итоге приходится изгаляться с соглашениями в духе «если второй параметр равен -1, значит на самом деле информации нет», которые не очевидны, забываемы и неудобны.

**Некуда вставить проверку валидности**

Давайте посмотрим на вот такую функцию, возвращающую диапазон рабочих температур некоторого устройства:

pair<int, int> SomeDevice::GetCelsiusTemperatureRange()

{

...

return make\_pair( -300, +30 );

}

В следствии опечатки в 1 символ функция (не сильно напрягаясь) расширила границы физической реальности, заявив что устройство может работать при -300 по Цельсию. Никакой проверки на валидность такой температуры ни в момент создания объекта пары, ни в момент возврата этого значения из функции попросту нет. И написать его вообще некуда.   
То ли дело, если бы возвращался объект диапазона температур, при создании которого можно было бы как-то поймать невалидное значение и отреагировать на него (ассерт, лог, исключение, замена на валидное значение и т.д.)

struct TemperatureRange

{

int minTemp;

int maxTemp;

TemperatureRange( int min, int max )

{

assert( min <= max );

assert( min >= -273 );

minTemp = min;

maxTemp = max;

}

}

TemperatureRange SomeDevice::GetCelsiusTemperatureRange()

{

...

return TemperatureRange( -300, +30 ); *// тут срабатывает assert!*

}

**Контрпример**

Что бы означало "**чаще всего плохо**" в названии статьи? Нужно признать, что иногда пары использовать можно и нужно. Например, у нас есть игра, в которой по ходу игровой механики для двух игроков нужно выбросить некоторые случайные значения (числа в диапазоне int). Это вполне может сделать функция вида:

pair<int, int> GetTwoRandomNumbers();

Почему эта функция не является плохой? Всё очень просто: 

* Чётко понятно, что находится в паре. Нет никаких способов двусмысленной трактовки.
* Порядок не имеет значения. Что сначала число для первого игрока, потом для второго, что наоборот — по барабану.
* Наша игра только на двух игроков и никогда (by design) не будет возможна для большего количества — беспокоиться о расширяемости не нужно
* Оба значения точно должны быть. Отсутствие одного из них невозможно.
* Проверка валидности не нужна — по определению весь диапазон int нам подходит.

Более того, в этом примере пара лучше отдельного класса (меньше кода), лучше out-параметров в виде указателей (не нужно проверять их на валидность) и лучше массива или вектора (те могут быть любого размера, что путает).   
В общем, пример имеет право на жизнь.

**Вывод**

Применение пар или кортежей мне кажется мало оправданным, если Вы пытаетесь писать понятный, легко читаемый и хорошо расширяемый код. Использование небольших классов или структур почти всегда даст выигрыш в читабельности, кроме совсем уж простых случаев.