Boost Concept Check Library (BCCL)

В чём смысл?

Все знают, что в C++ есть шаблоны. Все знают, что это очень удобная и мощная возможность языка, которая является одним из главных преимуществ C++ в мире языков программирования. Столь же мощными являются и ошибки компиляции с шаблонами, о которых в сообществе даже ходят легенды. Стоит попробовать инстанцировать шаблон с неподходящим типом (конечно же, по невнимательности, а не с целью каверзных экспериментов), и компилятор разражается ругательствами на несколько страниц далеко не самого увлекательного и захватывающего текста.

В качестве небольшого примера попробуйте скомпилировать следующий простейший код (нумерация строк убрана, чтобы код можно было просто скопировать в Ваш любимый компилятор):

```
#include <vector>
#include <complex>
#include <algorithm>

int main() {
    std::vector<std::complex<float> > v;
    std::stable_sort(v.begin(), v.end());
}
```

Если под рукой нет компилятора, или просто лень, можно воспользоваться онлайн-версиями:

http://cpp.sh/

https://www.onlinegdb.com/

https://wandbox.org/

Данный пример взят из документации **BCCL**, где называется он «Мотивирующий пример». Действительно, сообщение об ошибке сильно мотивирует ~~на то, чтобы никогда больше не использовать шаблоны~~ на то, чтобы попытаться как-то улучшить ситуацию. Именно для этого и создана библиотека **BCCL**. Пока на дворе не наступил 2020 год (концепты внесены в качестве предложения в C++20) приходится пользоваться несколько костыльными, но всё-таки решениями. По давно сложившейся традиции **Boost Concept Check Library** (повторил название, вдруг Вы уже забыли, что такое **BCCL**) использует макросы. Да-да, макросы, которые «не рекомендуется использовать». Иногда можно, ежели умеючи.

Попробуем сами

В папке **concepts** из материалов к заданию имеется набор исходников, которые призваны продемонстрировать полезность идеи концепций, а также показать, как эту саму идею (воплощённую в **BCCL**) использовать.

Motivation.cpp – пример, похожий на предыдущий, но с использованием самописной структуры **SomeStruct**. Раскомментируйте строчку 10:

```
//std::stable_sort(std::begin(values), std::end(values));
```

И попробуйте скомпилировать проект Motivation. Получится примерное такое же длинное сообщение об ошибке, как и примере выше.

BCCL_usage.cpp – пример использования **BCCL**. Снова самописная структура, и на этот раз не в одиночестве – также присутствует самописная функция сортировки (трудно представить, зачем нужно было бы писать свою функцию сортировки, которая вызывает библиотечную, но для небольшого примера – самое то). За счёт 10 строчки с использованием **BCCL**:

```
BOOST_CONCEPT_ASSERT((boost::LessThanComparable<T>));
```

сообщение об ошибке будет уже гораздо более понятным (по крайней мере, по мнению разработчиков библиотеки **BCCL**). Чтобы им восхититься, закомментируйте реализацию оператора меньше (строчки 7 – 9 включительно). Дополнительным бонусов станет автопереход по нажатию на ошибку компиляции (если используете IDE для сборки) не в потроха стандартной библиотеки (как было раньше), а на реализацию концепции, требования который не были удовлетворены. Причём, показывается именно содержательная строчка концепции – требование наличия оператора меньше:

```
require_boolean_expr(a < b);
```

Concept_creation.cpp – пример простейшей реализации концепции **LessThenComparable** и её использования. Снова по нажатию на ошибку компиляции (в нормальной IDE) курсор автоматически перейдёт в код концепции (только на этот раз уже написанной нами) в требование о наличии бинарного оператора меньше, возвращающего bool.

А теперь задание

Итак, домашнее задание по **BCCL** будет следующее:

1. Ответьте на каверзный вопрос – не проверяет ли реализованная нами концепция **LessThenComparableCustom** ещё и **DefaultConstructible** концепцию? Если да, то как этого избежать?
2. Воспользовавшись полученными знаниями, реализуйте концепцию **RandomAccessIterator** – итератор произвольного доступа. Подробное описание этой концепции можно найти здесь: http://en.cppreference.com/w/cpp/concept/RandomAccessIterator
3. Реализуйте набор тестов, который бы демонстрировал корректность реализации **RandomAccessIterator** концепции. Ошибка компиляции в негативных тестах вполне допустима, главное, чтобы в ней содержалась информации о том, что именно из концепции было нарушено.
4. Будьте аккуратны с названием класса для новой концепции. Концепция с названием **RandomAccessIterator** уже присутствует в **BCCL**. Поиск Кёнига? Самое время вспомнить.
5. Предложите ещё три концепции (придуманные, либо уже существующие в **BCCL**), которые могли бы облегчить Вам повседневное программирование.