TPL

Generated by Doxygen 1.8.3.1

Mon May 6 2013 15:38:32

Contents

Chapter 1

TPL: A preprocessor library for tuple manipulation

Introduction

This library provides preprocessor macros to manipulate tuples. Consider this simple example.

```
#include <iostream>
#include "TPL.h"

#define PRINT(x) std::cout << x

int main() {
    TPL_FOR_EACH(("Hello World ", 2013, '!'), PRINT);
}
```

Macro `TPL_FOR_EACH()` takes a tuple and a macro function and returns another tuple obtained by memberwise application of the given macro function. Hence, the line in `main()` first expands to

```
(PRINT("Hello World "), PRINT(2013), PRINT('!'));
```

and subsequently to

```
(std::cout << "Hello World ", std::cout << 2013, std::cout << '!');
```

Therefore, `Hello World 2013!` will appear on the screen.

A more interesting example is given in `enum.h`. It defines the macro `ENUM_IMPLEMENT()` that effortlessly create `enum`s and implements conversion functions to/from strings. For instance, this code

```
ENUM_IMPLEMENT(                    \
    color, default, (              \
        (red  , _, "Red"  , "R"), \
        (green, _, "Green", "G"), \
        (blue , _, "Blue" , "B")  \
    )                              \
)
```

creates a scoped `enum` named `color` with three enumerators: `red`, `green` and `blue`. The enumerators have default values, i.e. `0`, `1` and `2` respectively. Strings "Red" and "R" can be converted to red through `to_value()` and `color::red` can be converted to "Red" through `to_string()`. Similar facts hold for the other enumerators and their corresponding strings.

Definitions and conventions

TPL adopts the following terminology:

- A **sequence** is a collection of tokens separated by spaces. Example:

```
a b c
```

is a sequence with three elements: `a`, `b` and `c`.

- A **list** is a collection of tokens separated by commas. Example:

```
a, b, c
```

is a list with three elements: `a`, `b` and `c`.

- A **tuple** is a list surrounded by parenthesis. Example:

```
(a, b, c)
```

is a tuple with three elements: `a`, `b` and `c`.

- A **macro function** is a macro that acts similarly to a function and is expanded only if its name appears with a pair of parentheses after it. Examples:

```
#define MAX(a, b) (a>b?a:b) // MAX doesn't expand. MAX(1, 2) expands to 2.
#define ZERO() 0            // ZERO doesn't expand. ZERO() expands to 0.
```

contrast those with

```
#define ONE 1               // ONE expands to 1. This is not a macro function.
```

Some of macros have both tuple and list versions. In this case, the list version has a suffix `_L`. For instance, TPL_HEAD() yields the head of a tuple whereas `TPL_HEAD_L()` gives the head of a list:

```
TPL_HEAD((a, b, c)) // expands to a.
TPL_HEAD_L(a, b, c) // expands to a.
```

Supported compilers

TPL has been sucessfully tested with:

- GCC 4.7.2
- Clang 3.2
- Intel Compiler 13.1.0
- Visual Studio 2012

However, the example `enum.h` depends on some C++11 features (notably, scoped `enum`s) which the Intel Compiler 13.1.0 doesn't support.

Known issues

The maximum tuple and list size is 64: Most of TPL macros need to obtain the size of a tuple or list. (Failing to correctly perform this task yields undefined behaviour.) This is done through `TPL_SIZE_L()` and boils down to getting the number of arguments given to a variadic macro. Unfortunately, most preprocessors don't provide intrinsic support for this task. Although TPL can work around this, it's limitted to tuples and lists with at most 64 elements.

Recursion is emulated and limited to 64 levels: Macros don't support recursion and the preprocessor stops the expansion when it detects one. For instance, consider:

```
#define RECURSIVE(x) x, RECURSIVE(x + 1)
```

Then RECURSIVE(1) expands to

```
1, RECURSIVE(1 + 1)
```

This behavior does make sense: Had the expansion continued further, when should it stop? (Recall that there's no if inside a macro.)

The first consequence is that we can't nest a TPL macro inside itself. For instance, consider the following macros to get the square of the Euclidean norm of a tuple:

```
#define EUCLIDEAN_NORM_HELPER(x)  (x * x) +
#define EUCLIDEAN_NORM_SQR(v) TPL_FOR_EACH_S(v, EUCLIDEAN_NORM_HELPER) 0
```

Then, EUCLIDEAN_NORM_SQR((1, 2, 3)) expands to

```
(1 * 1) + (2 * 2) + (3 * 3) + 0
```

So far so good. Consider now that a matrix is defined by a tuple of tuples. A natural idea to get its norm squared would be

```
#define MATRIX_NORM_HELPER(v) EUCLIDEAN_NORM_SQR(v) +
#define MATRIX_NORM_SQR(m) TPL_FOR_EACH_S(m, MATRIX_NORM_HELPER) 0
```

Notice that MATRIX_NORM_SQR() calls TPL_FOR_EACH_S() which calls MATRIX_NORM_HELPER() which calls EUCLIDEAN_NORM_SQR() which calls TPL_FOR_EACH_S() again. Hence, the expansion will stops. Indeed, MATRIX_NORM_SQR(((1, 2, 3), (4, 5, 6))) expands to

```
TPL_FOR_EACH_S((1, 2, 3), EUCLIDEAN_NORM_HELPER ) 0 +
      TPL_FOR_EACH_S((4, 5, 6), EUCLIDEAN_NORM_HELPER) 0 + 0
```

Although recursion is impossible, it can be emulated by making a sequence of distinct macros calling each other as the following example illustrates.

```
#define RECURSIVE_3(x) x, RECURSIVE_2(x + 1)
#define RECURSIVE_2(x) x, RECURSIVE_1(x + 1)
#define RECURSIVE_1(x) x  // stops here
RECURSIVE_3(1)
```

The last line expands to

```
1, 1 + 1, 1 + 1 + 1
```

This emulation technique is behind a few TPL macros (e.g TPL_FOR_EACH()). The sequences of macros used to emulate recursion are define in file TPLRecursion.h. Each sequence is, obvioulsy, finite and this imposes a limit on the number of "recursion" levels which is 64.

Some macros are dangerous: As said above, most TPL macros depend on the proper functioning of TPL_SIZE_L() which in turn depends on the correctness of TPL_IS_EMPTY(). Surprinsingly, this macro's implementation (borrowed from Jens Gustedt) is not straightforward because the preprocessor considers an empty list of arguments as a list having one argument which is empty. Hence, distinghish lists with zero and one argument is very tricky. Although it works for most of cases TPL_IS_EMPTY() can still be fooled by some dangerous macros whose expansions start with parenthesis. For instance

```
#define FOOLISH() (
TPL_IS_EMPTY_L(FOOLISH)
```

yields the following error on the GNU preprocessor:

> unterminated argument list invoking macro "TPL_HAS_COMMA_L_A"

Despite the diagnostics, the preprocessor carries on and expands the macro to 0.

A worse example is this

```
#define NOT_EMPTY (a)
TPL_IS_EMPTY_L(NOT_EMPTY)
```

which silently (no error is reported) expands to 1.

Related work

Boost.Preprocessor is another library which provides preprocessor macros to manipulate tuples and other data types. Specifically for tuples, it provides two families of functions: *variadic* and *non variadic*. Non variadic functions require the user to pass the size of the tuple as an argument. This is annoying and error prone. Variadic functions can deduce the size of the tuple but apparently don't work properly with zero-sized tuples as the next example shows:

```
#define BOOST_PP_VARIADICS
#include <boost/preprocessor/tuple/size.hpp>

BOOST_PP_TUPLE_SIZE((a)) // OK: expands to 1.
BOOST_PP_TUPLE_SIZE(())  // Wrong: also expands to 1.
```

Being able to work with zero-sized tuples wasn't the main motivation for writting TPL though. I've wanted to learn more about preprocessor metaprogramming techniques and TPL was an exercise that I set myself to. Others can benefit from studing the code which is self-contained (two header files) and, IMHO, much simpler to understand than Boost's.

Makefile tags

The `makefile` shipped with TPL can run tests and build an example. More precisely, the available tags are described below.

- **test**

  Run the preprocessor on `test.cpp` and `diff` the result against `expect.txt`. If they don't match, it means that the test have failed and the differences will be shown.

- **color**

  Build the executable `color` which tests macro `ENUM_IMPLEMENT()` defined in `enum.h`.

- **recursion**

  Build the executable `recursion` which is used to generate `TPLRecursion.h`. You don't need this unless you want to change the maximum allowed tuple size (currently `64`).

- **clean**

  Remove `color`, `recursion` and `results.txt`.

Files

The following describes the files in this repository. To use the library you only need `TPL.h` and `TPLRecursion.-h`.

- `color.h`, `color.cpp` and `color2.cpp`

  Test macro `ENUM_IMPLEMENT()` defined in `enum.h`. You can build the executable with `make color`.

- `doxyfile`

  Used to generate the documentation.

- `enum.h`

  This is an example built on top of TPL. It defines macro `ENUM_IMPLEMENT()` which effortlessly implements `enum`s and conversion functions to/from string.

- `expected.txt`

  Expected output from preprocessing `test.cpp`. You can run the tests with `make test`. This will produce a file `results.txt` which is compared with `expected.txt`.

- `makefile`

  To run the test, build the example and do other tasks.

- `README.md`

  Main documentation page. (The page that you're reading right now.)

- `recursion.cpp`

  This program helps to generate `TPLRecursion.h`.

- `test.cpp`

  Unit tests for TPL. This file is not meant to be compiled but only preprocessed. You can use `make test` for this task.

- `TPL.h` and `TPLRecursion.h`

  Implemention of TPL macros.

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Module Documentation

4.1 Create enums and conversion functions with TPL.

The file enum.h implements the macro ENUM_IMPLEMENT() which takes a tuple containing data for defining the `enum`, its enumerators and strings which the enumerators can be converted to and from.

Chapter 5

File Documentation

5.1 enum.h File Reference

Definition of `ENUM_IMPLEMENT()`.

Macros

- #define ENUM_IMPLEMENT(name, flag, data)

  Define an `enum` *and implement conversion to_string() and to_value().*

Functions

- const char ∗ to_string (name val)

  Convert an enumerator into a string.
- void to_string_error_handler (const char ∗name, int val)

  Handler of to_string() failures.
- template<typename name >
  name to_value (const char ∗str)

  Convert a string into an enumerator.
- void to_value_error_handler (const char ∗name, const char ∗str, const char ∗strs[ ], std::size_t n)

  Handler of to_value() failures.

5.1.1 Detailed Description

Definition of `ENUM_IMPLEMENT()`.

5.1.2 Macro Definition Documentation

5.1.2.1 #define ENUM_IMPLEMENT(*name, flag, data*)

Define an `enum` and implement conversion to_string() and to_value().

This is an example of how TPL macros can help metaprogramming. Calling this macro passing in the data that `enum`s are based on (name, enumerator, values and strings) will effortlessly define the `enum` and implement the conversion functions to_string() and to_value().

This macro takes three arguments: the `name` of the `enum`, the `flag` that specifies if enumerator values are default or user provided and a tuple `data` containing enumerators data.

`flag` must be either `default` or `user`. In the former case, enumerators will take default values: `0`, `1`, `2`, etc. In the latter case the values are provided by the user in `data`.

Each element of `data` is a tuple containing the data associated with each enumerator. This tuple's first element is the enumerator name. If `flag` is `user`, then the second element is the value of the enumerator. If `flag` is `default`, then the second element is ignored and the enumerator is given a default value. The remaining elements of the tuple, from the third onwards, are literal strings which the enumerator converts to and from. Any such string can be converted to the enumerator through to_value(). Conversely, to_string() converts the enumerator to its *main string*, that is, the first string provided for the enumerator (i.e. the third element of the tuple).

Example:

```
ENUM_IMPLEMENT(                         \
    color, default, (                   \
        (red  , _, "Red"  , "R"),       \
        (green, _, "Green", "G"),       \
        (blue , _, "Blue" , "B")        \
    )                                   \
)
```

This creates in the current scope (global, `namespace` or `class`) a scoped `enum` named `color` with three enumerators: `red`, `green` and `blue`. The enumerators have default values, i.e. `0`, `1` and `2` respectively.

In the same scope, it implements conversion functions `to_value()` and `to_string()`. The former can convert `"Red"` and `"R"` to `color::red` whereas the latter converts `color::red` to `"Red"`. Similar conversions apply between other enumerators and their associated strings.

If instead of default values, you want to set `color::red = 1`, `color::green = 2` and `color::blue = 4`, then the following should be used:

```
ENUM_IMPLEMENT(                         \
    color, user, (                      \
        (red  , 1, "Red"  , "R"),       \
        (green, 2, "Green", "G"),       \
        (blue , 4, "Blue" , "B")        \
    )                                   \
)
```

Parameters

name	The `enum` name.
flag	Either `default` or `user`, indicates if enumerators take default or user provided values.
data	Tuple containing data of each enumerator.

Returns

Code defining the `name`, to_string() and to_value().

5.1.3 Function Documentation

5.1.3.1 const char∗ to_string (name *val*)

Convert an enumerator into a string.

If the conversion fails, then to_string_error_handler() is called to handle the error. Notice that a failure in to_string() is probably caused by a bug. Indeed, the given argument is generally a valid enumerator unless it's obtained from casting an invalid integer value to the `enum name`.

Example:

```
ENUM_IMPLEMENT(                         \
```

```
    color, default, (                  \
        (red  , _, "Red"  , "R"), \
        (green, _, "Green", "G"), \
        (blue , _, "Blue" , "B")  \
    )                                  \
)

assert(to_string(color::red  ) == std::string("Red"  ));
assert(to_string(color::green) == std::string("Green"));
assert(to_string(color::blue ) == std::string("Blue" ));

assert(to_string(color::red  ) != std::string("R"));
assert(to_string(color::green) != std::string("G"));
assert(to_string(color::blue ) != std::string("B"));
```

Parameters

val	The enumerator to be converted.

Returns

The string.

5.1.3.2 void to_string_error_handler (const char * *name,* int *val*)

Handler of to_string() failures.

This function is called by to_string() to handle conversion errors. Only a declaration is provided here and users must provide the implementation.

A failure in to_string() is probably due to a bug. Therefore, consider calling std::terminate() in the implementation.

Parameters

name	The stringfied name of the `enum`.
val	The value of the enumerator that could not be converted.

5.1.3.3 template<typename name > name to_value (const char * *str*)

Convert a string into an enumerator.

If the conversion fails, then to_value_error_handler() is called to handle the error.

Example:

```
ENUM_IMPLEMENT(                        \
    color, default, (                  \
        (red  , _, "Red"  , "R"), \
        (green, _, "Green", "G"), \
        (blue , _, "Blue" , "B")  \
    )                                  \
)

assert(color::red   == to_value<color>("Red"  ));
assert(color::green == to_value<color>("Green"));
assert(color::blue  == to_value<color>("Blue" ));

assert(color::Red   == to_value<color>("R"));
```

```
assert(color::Green == to_value<color>("G"));
assert(color::Blue  == to_value<color>("B"));
```

Template Parameters

name	The `enum` name.

Parameters

str	The string to be converted.

Returns

The eenum_handlersnumerator.

5.1.3.4 void to_value_error_handler (const char ∗ *name,* const char ∗ *str,* const char ∗ *strs[ ],* std::size_t *n*)

Handler of to_value() failures.

This function is called by to_value() to handle conversion errors. Only a declaration is provided here and users must provide the implementation.

Parameters

name	The stringfied name of the `enum`.
str	The string that failed to be converted.
strs	The complete list of allowed strings.
n	The size of strs.

5.2 TPL.h File Reference

Definition of TPL macros library.

Macros

- #define TPL_TO_LIST(t)

  Converts tuple to list.
- #define TPL_IS_EMPTY(t)

  Test if a tuple is empty or not.
- #define TPL_IS_EMPTY_L(...)

  Test if a list is empty or not.
- #define TPL_SIZE(t)

  Get the size of a tuple.
- #define TPL_SIZE_L(...)

  Gets the size of a list.
- #define TPL_HEAD(t)

  Get the head of a tuple.
- #define TPL_HEAD_L(...)

  Get the head of a list.
- #define TPL_TAIL(t)

  Get the tail of a tuple.
- #define TPL_TAIL_L(...)

Get the tail of a list.
- #define TPL_ELEMENT(t, i)

    Extract the i-th (zero-based) element of a tuple.
- #define TPL_APPEND(t, x)

    Append an element to a tuple.
- #define TPL_INSERT(t, x)

    Insert an element to the begining of a tuple.
- #define TPL_ROTATE(t)

    Rotate a tuple to the left.
- #define TPL_RECURSE(t, n, F)

    Recursively apply a function to a tuple.
- #define TPL_FOR_EACH(t, F, a)

    Get a tuple obtained by memberwise application of a function.
- #define TPL_FOR_EACH_L(t, F, a)

    Get a list obtained by memberwise application of a function.
- #define TPL_FOR_EACH_S(t, F, a)

    Get a sequence obtained by memberwise application of a function.

5.2.1 Detailed Description

Definition of TPL macros library.

5.2.2 Macro Definition Documentation

5.2.2.1 #define TPL_APPEND(*t*, *x*)

Append an element to a tuple.

Example: `TPL_APPEND((a, b), c)` expands to `(a, b, c)`.

Parameters

t	The tuple.
x	The element.

Returns

The appended tuple.

5.2.2.2 #define TPL_ELEMENT(*t*, *i*)

Extract the *i-th* (zero-based) element of a tuple.

Example: `TPL_ELEM((a, b, c), 1)` expands to `b`. ∗up

Parameters

t	The tuple.
i	The index of the element to be extracted.

Returns

The i-th element of *t*.

5.2.2.3 #define TPL_FOR_EACH(*t*, *F*, *a*)

Get a tuple obtained by memberwise application of a function.

Given a tuple `t = (x0, ..., xn)` and a function `F` taking two arguments. `TPL_FOR_EACH(t, F, a)` expands to `(F(x0, a), ..., F(xn, a))`.

Notice that the output is a tuple. For similar functions that return a list or a sequence, see `TPL_FOR_EACH_L` and `TPL_FOR_EACH_S`.

Parameters

t	The tuple.
F	The function.
a	Extra argument for F `F`.

Returns

A tuple obtained by memberwise application of `F` to `t`.

5.2.2.4 #define TPL_FOR_EACH_L(*t*, *F*, *a*)

Get a list obtained by memberwise application of a function.

Given a tuple `t = (x0, ..., xn)` and a function `F` taking two arguments. `TPL_FOR_EACH_L(t, F, a)` expands to `F(x0, a), ..., F(xn, a)`.

Notice that the output is a list. For similar functions that return a tuple or a sequence, see `TPL_FOR_EACH` and `TPL_FOR_EACH_S`.

Parameters

t	The tuple to be iterated over.
F	The function.
a	Extra argument for F `F`.

Returns

A list obtained by memberwise application of `F` to `t`.

5.2.2.5 #define TPL_FOR_EACH_S(*t*, *F*, *a*)

Get a sequence obtained by memberwise application of a function.

Given a tuple `t = (x0, ..., xn)` and a function `F` taking two arguments. `TPL_FOR_EACH_S(t, F, a)` expands to `F(x0, a) ... F(xn, a)`.

Notice that the output is a sequence. For similar functions that return a tuple or a list, see `TPL_FOR_EACH` and `TPL_FOR_EACH_L`.

Parameters

t	The tuple to be iterated over.
F	The function.
a	Extra argument for F `F`.

Returns

A sequence obtained by memberwise application of `F` to `t`.

5.2.2.6 #define TPL_HEAD(*t*)

Get the head of a tuple.

Example: `TPL_HEAD((a, b, c))` expands to `a`.

t The tuple.

Returns

The head.

5.2.2.7 #define TPL_HEAD_L(...)

Get the head of a list.

Example: `TPL_HEAD_L(a, b, c)` expands to `a`.

Parameters

...	The list.

Returns

The head.

5.2.2.8 #define TPL_INSERT(*t*, *x*)

Insert an element to the begining of a tuple.

Example: `TPL_INSERT((a, b), c)` expands to `(c, a, b)`.

Parameters

t	The tuple.
x	The element.

Returns

The inserted tuple.

5.2.2.9 #define TPL_IS_EMPTY(*t*)

Test if a tuple is empty or not.

Example: `TPL_IS_EMPTY(())` expands to `1` and `TPL_IS_EMPTY((a, b, c))` expands to `0`.

Parameters

t	The tuple.

Returns

1 if the tuple is empty and 0 otherwise.

5.2.2.10 #define TPL_IS_EMPTY_L(...)

Test if a list is empty or not.

Example: `TPL_IS_EMPTY_L()` expands to `1` and `TPL_IS_EMPTY_L(a, b, c)` expands to `0`.

Parameters

...	The list.

Returns

1 if the list is empty and 0 otherwise.

5.2.2.11 #define TPL_RECURSE(*t*, *n*, *F*)

Recursively apply a function to a tuple.

The function takes one tuple parameter and returns a tuple.

Example: `TPL_RECURSE(t, 3, F)` expands to `F(F(F(t)))`.

Parameters

t	The tuple.
n	The number of times the function is iterated.
F	The function.

Returns

The iteration of F.

5.2.2.12 #define TPL_ROTATE(*t*)

Rotate a tuple to the left.

Example: `TPL_ROTATE((a, b, c))` expands to `(b, c, a)`.

Parameters

t	The tuple.

Returns

The rotated tuple.

5.2.2.13 #define TPL_SIZE(*t*)

Get the size of a tuple.

Example: `TPL_SIZE((a, b, c))` expands to `3`.

Parameters

t	The tuple.

Returns

The size of t.

5.2.2.14 #define TPL_SIZE_L(...)

Gets the size of a list.

Example: `TPL_SIZE_L(a, b, c)` expands to `3`.

Precondition

The number of elements cannot exceed `64`.

Parameters

...	The list.

Returns

The size.

5.2.2.15 #define TPL_TAIL(*t*)

Get the tail of a tuple.

Example: `TPL_TAIL_L((a, b, c))` expands to `(b, c)`.

Parameters

t	The tuple.

Returns

The tail.

5.2.2.16 #define TPL_TAIL_L(...)

Get the tail of a list.

Example: `TPL_TAIL_L(a, b, c)expands to b, c.`

Parameters

`...`	`The list.`

Returns

`The tail.`

5.2.2.17 #define TPL_TO_LIST(*t*)

Converts tuple to list.

Example: `TPL_TO_LIST((a, b, c))` expands to `a, b, c`

Parameters

t	The tuple.

Returns

The list.

Index