Pump is **U**seful for **M**eta **P**rogramming.

The Problem

Template and macro libraries often need to define many classes, functions, or macros that vary only (or almost only) in the number of arguments they take. It's a lot of repetitive, mechanical, and error-prone work.

Variadic templates and variadic macros can alleviate the problem. However, while both are being considered by the C++ committee, neither is in the standard yet or widely supported by compilers. Thus they are often not a good choice, especially when your code needs to be portable. And their capabilities are still limited.

As a result, authors of such libraries often have to write scripts to generate their implementation. However, our experience is that it's tedious to write such scripts, which tend to reflect the structure of the generated code poorly and are often hard to read and edit. For example, a small change needed in the generated code may require some non-intuitive, non-trivial changes in the script. This is especially painful when experimenting with the code.

Our Solution

Pump (for Pump is Useful for Meta Programming, Pretty Useful for Meta Programming, or Practical Utility for Meta Programming, whichever you prefer) is a simple meta-programming tool for C++. The idea is that a programmer writes a `foo.pump` file which contains C++ code plus meta code that manipulates the C++ code. The meta code can handle iterations over a range, nested iterations, local meta variable definitions, simple arithmetic, and conditional expressions. You can view it as a small Domain-Specific Language. The meta language is designed to be non-intrusive (s.t. it won't confuse Emacs' C++ mode, for example) and concise, making Pump code intuitive and easy to maintain.

Highlights

- The implementation is in a single Python script and thus ultra portable: no build or installation is needed and it works cross platforms.
- Pump tries to be smart with respect to Google's style guide: it breaks long lines (easy to have when they are generated) at acceptable places to fit within 80 columns and indent the continuation lines correctly.
- The format is human-readable and more concise than XML.
- The format works relatively well with Emacs' C++ mode.

Examples

The following Pump code (where meta keywords start with `$`, `[[` and `]]` are meta brackets, and `$$` starts a meta comment that ends with the line):

```
$var n = 3     $$ Defines a meta variable n.
$range i 0..n  $$ Declares the range of meta iterator i (inclusive).
$for i [[
               $$ Meta loop.
// Foo$i does blah for $i-ary predicates.
$range j 1..i
template <size_t N $for j [[, typename A$j]]>
class Foo$i {
$if i == 0 [[
  blah a;
]] $elif i <= 2 [[
```

```
  blah b;
]] $else [[
  blah c;
]]
};

]]
```

will be translated by the Pump compiler to:

```
// Foo0 does blah for 0-ary predicates.
template <size_t N>
class Foo0 {
  blah a;
};

// Foo1 does blah for 1-ary predicates.
template <size_t N, typename A1>
class Foo1 {
  blah b;
};

// Foo2 does blah for 2-ary predicates.
template <size_t N, typename A1, typename A2>
class Foo2 {
  blah b;
};

// Foo3 does blah for 3-ary predicates.
template <size_t N, typename A1, typename A2, typename A3>
class Foo3 {
  blah c;
};
```

In another example,

```
$range i 1..n
Func($for i + [[a$i]]);
$$ The text between i and [[ is the separator between iterations.
```

will generate one of the following lines (without the comments), depending on the value of `n`:

```
Func();              // If n is 0.
Func(a1);            // If n is 1.
Func(a1 + a2);       // If n is 2.
Func(a1 + a2 + a3);  // If n is 3.
// And so on...
```

Constructs

We support the following meta programming constructs:

`$var id = exp`	**Defines a named constant value. `$id` is valid util the end of the current meta lexical block.**
$range id exp..exp	Sets the range of an iteration variable, which can be reused in multiple loops later.
$for id sep [code]	Iteration. The range of `id` must have been defined earlier. `$id` is valid in `code`.
`$($)`	Generates a single `$` character.
`$id`	Value of the named constant or iteration variable.
`$(exp)`	Value of the expression.
`$if exp [[ code ]] else_branch`	Conditional.
`[[ code ]]`	Meta lexical block.
`cpp_code`	Raw C++ code.
`$$ comment`	Meta comment.

Note: To give the user some freedom in formatting the Pump source code, Pump ignores a new-line character if it's right after `$for foo` or next to `[[` or `]]`. Without this rule you'll often be forced to write very long lines to get the desired output. Therefore sometimes you may need to insert an extra new-line in such places for a new-line to show up in your output.

Grammar

```
code ::= atomic_code*
atomic_code ::= $var id = exp
    | $var id = [[ code ]]
    | $range id exp..exp
    | $for id sep [[ code ]]
    | $($)
    | $id
    | $(exp)
    | $if exp [[ code ]] else_branch
    | [[ code ]]
    | cpp_code
sep ::= cpp_code | empty_string
else_branch ::= $else [[ code ]]
    | $elif exp [[ code ]] else_branch
    | empty_string
exp ::= simple_expression_in_Python_syntax
```

Code

You can find the source code of Pump in scripts/pump.py. It is still very unpolished and lacks automated tests, although it has been successfully used many times. If you find a chance to use it in your project, please let us know what you think! We also welcome help on improving Pump.

Real Examples

You can find real-world applications of Pump in Google Test and Google Mock. The source file `foo.h.pump` generates `foo.h`.

Tips

- If a meta variable is followed by a letter or digit, you can separate them using `[[]]`, which inserts an empty string. For example `Foo$j[[]]Helper` generate `Foo1Helper` when `j` is 1.
- To avoid extra-long Pump source lines, you can break a line anywhere you want by inserting `[[]]` followed by a new line. Since any new-line character next to `[[` or `]]` is ignored, the generated code won't contain this new line.