Chapter 1

Expressing C++ Template Metaprograms as Lambda expressions

Ábel Sinkovics[1], Zoltán Porkoláb[2]
Category: Research Paper

Abstract: Template metaprogramming is an emerging new direction of generative programming: with clever definitions of templates we can enforce the C++ compiler to execute algorithms at compilation time. Although the relationship between template metaprograms and functional programming is well-known, there are no studies to reveal how substantial features of functional programming can be implemented in the means of C++ template metaprograms. In this paper we overview the most essential elements of functional programming: lazy and eager evaluations, lazy data types, currying, fixpoint operation, etc. and show their possible implementations with metaprograms. For this purpose we define and implement a translator to map lambda expressions to C++ template metaprograms. Using the tool lambda expressions embedded into C++ host language and expressed by their natural syntax are translated to native C++ code. As current C++ metaprograms are mostly written using the intricate syntax of templates, this study is also intended as a further step to implement a more understandable and maintainable functional style interface for C++ template metaprograms.

[1]Eötvös Loránd University, Faculty of Informatics, Pázmány Péter sétány 1/C, 1117 Budapest, Hungary; `abel@elte.hu`

[2]Eötvös Loránd University, Faculty of Informatics, Pázmány Péter sétány 1/C, 1117 Budapest, Hungary; `gsd@elte.hu`

1.1 INTRODUCTION

In 1994 Erwin Unruh wrote a program in standard C++ which didn't compile, but the error messages the compiler displayed contained a list of prime numbers. He used C++ templates and the template instantiation rules to write a program that is executed as a side effect of compilation.

Abrahams and Gurtovoy [1] defined the term template metafunction as a special template class: the arguments of the metafunction are the template parameters of the class, the value of the function is a nested type of the template called `type`. For example:

```
template <class T>
struct makeConst {
  typedef const T type;
};
```

The example defined a metafunction called `makeConst` taking a class as an argument. The result is another class which is a nested type of the metafunction. In the example above the function can be called with an `int` argument in the following way: `makeConst<int>::type`.

Data (such as integral constants) can be expressed in template metaprograms as well. Templates can have integral constant arguments and the value of a template metafunction can be defined as a static constant. Here is an example of addition:

```
template <int a, int b>
struct Plus {
  static const int value = a + b;
};
```

Having defined this the expression `Plus<6, 7>::value` evaluates to the 13 constant (at compile time).

1.1.1 Connection between C++ template metaprogramming and functional programming

Template metaprograms are functional programs executed at compile time via template instantiation. It supports the most important features of functional programs: higher order functions, lazy evaluation, pattern matching, recursion.

Abrahams and Gurtovoy [1] defined metafunction classes, which are classes with a nested metafunction called `apply`. Higher order functions can be represented using metafunction classes: since a metafunction class is a class, it can be the result (or an argument) of a metafunction. Here is an example of a metafunction class:

```
struct MakeConst {
  template <class T>
  struct apply {
    typedef const T type;
  };
};
```

Evaluation of metaprograms happens via instantiation of templates, and templates are instantiated by need only [3] making lazy evaluation possible. For example the `if` structure [1] can be implemented by a metafunction taking a condition and two classes as values. Here is an example:

```
  template <bool cond, class T, class F>
  struct If {
    typedef T type;
  };

  template <class T, class F>
  struct If<false, T, F> {
    typedef F type;
  };
```

In this example only one of `T` and `F` is instantiated when the metafunction is evaluated (unless `T` and `F` are defined explicitly somewhere else).

Template metafunctions support pattern matching: specialisations (based on the template arguments) of template classes can be defined. For example:

```
  template <class T>
  struct removeConst {
    typedef T type;
  };

  template <class T>
  struct removeConst<const T> {
    typedef T type;
  };
```

This metafunction removes constness of a type using specialisation.

Recursive template metafunctions can be defined as well because a class is in scope in it's definition. Recursion can be stopped by pattern matching. Here is an example:

```
  template <unsigned int n>
  struct factorial {
    static const unsigned int
      value = n * factorial<n-1>::value;
  };
```

```
template <>
struct factorial <0> {
  static const unsigned int value = 1;
};
```

This metafunction calculates the factorial of a number;

1.1.2 Maintenance problems with template metaprograms

We have seen what template metaprogramming is capable of, but it has drawbacks as well. C++ wasn't designed to support template metaprogramming, this capability of the language was discovered later. Because of this, template metaprogramming is not a simple and easy to use tool. The syntax is intricate and error messages displayed by the C++ compilers are difficult to read and understand. Having tools supporting development of template metaprograms could let developers safely use them in production software.

We examine how functional languages could be used to write template metaprograms in, letting developers use a better syntax for writing and maintaining metaprograms. Since lambda expressions are capable of expressing any functional program we show how lambda expressions can be used to express C++ template metaprograms in. We wrote a translator which can translate nested lambda expressions into template metaprograms in C++ code.

1.2 FUNCTIONAL FEATURES

We use the definition of non-typed enriched lambda expressions from [21]. The only character which we decided to change was the λ character which we replaced by a $\backslash$ character.

```
<expression> ::=
  <constant> | <variable> |
  <expression> <expression> |
  \ <variable> . <expression> |
  ( <expression> )
```

Decimal numbers and built-in operators are valid constants. Supported operators are: $+$, $-$, $*$, $/$, %, $<$, $>$, $<=$, $>=$, $<>$, $=$, \$. (% is modulo and \$ is the fixpoint operator). We restrict the form of a general lambda abstraction allowing only one variable, i.e. the expression `\xy.E` should be written in form of `\x.\y.E`. This restriction doesn't affect expressiveness.

We have defined a conversion of these expressions to C++ template metaprograms. During the execution of those metaprograms the C++ compiler builds the graph of the expression and reduces it lazily. Lambda expressions can be embedded into C++ code with the following syntax:

```
__lambda <name> = <expression>;
```

Our compiler compiles them into C++ classes (metafunction classes [1]) implementing the lambda expression. The names of the classes are the names of the lambda expressions indicating that names have to be valid C++ names. Since these expressions are translated into C++ classes they can be at any part of the code where classes can be defined [3].

1.2.1 Lazy and eager evaluation

Our compiler supports lazy evaluation of lambda expressions: every (sub)expression is evaluated only when it's value is needed. It makes implementation of infinite data structures (such as infinite lists) possible. Eager evaluation is supported by the classes implementing the lambda expressions in C++ but are not supported directly in the lambda expressions themselves: they are always evaluated lazily.

1.2.2 Lazy data types

Since lambda expressions are capable of expressing lazy data types, they can be represented in C++ template metaprograms using our compiler. For example lazy lists can be expressed:

```
__lambda true = \x.\y. x;
__lambda false = \x.\y. y;

__lambda pair = \x.\y.\z. z x y;
__lambda first = \x. x true;
__lambda second = \x. x false;

__lambda cons = \x.\y. pair false (pair x y);
__lambda nil = pair true true;
__lambda head = \x. first (second x);
__lambda tail = \x. second (second x);
```

as it is defined in [5], they are lambda expressions representing lists.

1.2.3 Currying

Currying is supported: when the number of elements applied to a function symbol is less than the number of elements required by the function symbol the result is a new function symbol. For example: we have an anonymous function requiring two elements to be applied to it: `\x.\y. + x y`. When only one element is applied to this function the result is a new function requiring one element to be applied to it. `(\x.\y. + x y) 5` is equivalent to `\y. + 5 y`.

The C++ template metaprogram equivalent of these lambda expressions supports currying as well. Currying has to be used explicitly: only one element can be applied to the metaprogramming equivalent of a function at a time. Applying one element to the equivalent of a function requiring multiple elements being applied to it is evaluated to the equivalent of another function requiring less (by one) elements. Another element needs to be applied to that function after that, etc. The same thing happens in lambda expressions in a series of applications, for example in `f 5 8`.

1.2.4 Interoperability with native C++ metafunctions

Lambda expressions have C++ equivalents and they can be implemented natively as well. Natively implemented lambda expressions can be used in lambda expressions (as constants). For example:

```
struct NativeLambdaExpression {
  // native implementation...
};

__lambda f = \n. NativeLambdaExpression 2 n;
```

It makes extension of the built-in operators possible and parts of the expressions can be implemented using other techniques.

Lambda expressions can be used by native C++ template metaprograms as well since lambda expressions are compiled into template metaprograms. After they are compiled into template metaprograms there is no difference between a natively implemented lambda expression and a compiled one: the compiled one can be used as a natively implemented one. Lambda expressions can be used as built-in functions in other lambda expressions, for example:

```
__lambda add = \a.\b. + a b;
__lambda f = \n. * n (add 6 7);
```

Lambda expressions can be used in their own definition simplifying the creation of recursive expressions:

```
__lambda rec = \n. (< n 1) 13 (rec (- n 1));
```

The lambda definitions behave similarly to the definitions of a `letrec` block in Peyton Jones's book [21]. Here is a `letrec` block and it's equivalent in our solution:

```
letrec
  add = \a.\b. + a b
  f = \n. * n (add 6 7)
  rec = \n. (< n 1) 13 (rec (- n 1))
in
  (some expression....)
```

```
__lambda add = \a.\b. + a b;
__lambda f = \n. * n (add 6 7);
__lambda rec = \n. (< n 1) 13 (rec (- n 1));
// some C++ code containing lambda expression(s)...
```

Due to the visibility rules of C++ [3] lambda expressions are visible after their declaration. For example the following code wouldn't compile because `b` is defined after `a`:

```
__lambda a = \n. b n;
__lambda b = \n. + 1 n;
```

Our compiler supports forward declaration of lambda expressions by ensuring that every lambda expression compiled to C++ will be implemeneted as a `struct`. The previous example `b` can be declared before `a`:

```
struct b;
  __lambda a = \n. b n;
  __lambda b = \n. + 1 n;
```

1.2.5 Using lambda expressions from other lambda expressions

A lambda expression embedded in a C++ code gives a name to a lambda expression:

```
__lambda f = \n. * (+ n 3) 2;
```

This is translated into a template metafunction class called `f` taking one argument

1.2.6 Fixpoint operation and recursion

The fixpoint operator can be expressed as a regular lambda expression, our compiler can compile it into a C++ template metaprogram, but (for more efficient programs) our compiler provides it as a built-in operator: $. This is the `Y` operator from [6]: for every `H` lambda expression:

```
$ H = H ($ H)
```

It can be used to implement recursion, but lambda expressions we compile can reference themselves as well:

```
__lambda factorial =
  \n. (= n 0) 1 (* n (factorial (- n 1)));
```

Our compiler generates the following code from this example:

```
struct factorial;

struct factorial__implementation
{
  template <class n>
  struct apply
  {
    typedef
      lambda::Application<
        lambda::Application<
          lambda::Application<
            lambda::Application<
              lambda::OperatorEquals,
              n
            >,
            lambda::Constant<int, 0>
          >,
          lambda::Constant<int, 1>
        >,
        lambda::Application<
          lambda::Application<
            lambda::OperatorMultiply,
            n
          >,
          lambda::Application<
            factorial,
            lambda::Application<
              lambda::Application<
                lambda::OperatorMinus,
                n
              >,
              lambda::Constant<int, 1>
            >
          >
        >
      >
      type;
  };
};

struct factorial : factorial__implementation
{
  typedef factorial__implementation base;
};
```

1.3 IMPLEMENTATION DETAILS

We define our C++ implementation of the elements of lambda expressions.

1.3.1 Constants

Constants are implemented by a class. There are two types of constants: integral constants and types. Types are implemented by themselves, for example the type `int` is implemented by `int`. Integral constants are implemented by a wrapper class, such as the wrappers from `boost::mpl` [1].

1.3.2 Lambda abstractions

Lambda abstractions are implemented by metafunction classes [1] whose embedded `apply` metafunction takes exactly one argument. The name of the argument is the name of the variable the lambda abstraction bounds.

For example here is a lambda expression and it's implementation:

```
// The lambda expression
__lambda I = \x. y;

// It's implementation
struct I {
  template <class x>
  struct apply {
    typedef y type;
  };
};
```

1.3.3 Variables

Variables are implemented by their name. A name symbol from the lambda expression becomes a name symbol in C++. Binding of the names in lambda abstractions is done by the C++ compiler. As we could see it in the previous example the lambda expression `y` becomes `typedef y type` in the C++ template metaprogram. The example has a lambda abstraction binding `x`. This lambda abstraction is represented by a template metafunction taking one argument called `x`. When this metafunction is instantiated the `x` symbols in it's body (if there are any) are replaced by the class the metafunction is instantiated with.

1.3.4 Eagerly evaluated applications

Eager application of a lambda expression to a lambda abstraction is implemented by the evaluation of the `apply` metafunction. The C++ compiler does the β conversion during the instantiation because the name of the bounded variable is the name of the argument of the nested `apply` metafunction (and the variables are implemented by their names).

The `I` lambda expression defined in the previous section can be evaluated either in an eager or lazy way. To specify eager evaluation, the user should use the following C++ construct:

```
typedef I::apply<I>::type ApplicationOfIToItself;
```

We will discuss lazy evaluation in subsection 1.3.6.

1.3.5 Currying in built-in functions

Built-in in functions (such as the arithmetical or logical operators) have more than one arguments. Their implementation has to support currying. They have to be implemented as a lambda abstraction. For example applying an element on the plus operator has to evaluate to another lambda abstraction, applying another element on that has to evaluate to a constant (and the value of it has to be the sum of the arguments). It can be implemented easily using nested types and templates. As an example here is the implementation of the plus operator:

```
struct OperatorPlus {
  template <class a>
  struct apply {
    struct type {
      template <class b>
      struct apply {
        // ... native implementation of addition,
        // possibly by boost::mpl
      };
    };
  };
}
```

We assume that every built-in function supports partial evaluation (to a lambda abstraction).

1.3.6 Lazy application

Applications in lambda expressions are evaluated only when their value is needed, they can't be translated into eager applications. We use the following template to implement lazy application:

```
template <class left, class right>
struct Application {};
```

Using this template expressions for lazy evaluation can be built as binary trees of applications: the instances of the `Application` template represent the application nodes of the tree, the `left` and `right` arguments represent the sub trees of the application nodes.

We define a metafunction implementing reduction of expressions to weak head normal form [5]. Stand alone lambda abstractions, constants and built-in functions are in weak head normal form. Lazy applications are never in weak head normal form, since we assume that every built-in function supports partial evaluation. These considerations simplify the reduction algorithm:

```
while (the top level element is a lazy application)
  reduce the left side of the top level element to
    weak head normal form
  evaluate the top level application
```

We implemented this in a metafunction called `Reduce`:

```
template <class T> struct Reduce { typedef T type; };

template <class left, class right>
struct Reduce< Application<left, right> > {
  typedef
    typename Reduce<
      typename
      Reduce<left>::type::template
      apply<right>::type
    >::type  type;
};
```

The general case handles lambda expressions which are already in weak head normal form, there is a specialisation of the template for reducing lazy applications in normal order reduction: it reduces the left sub-expression of the application to weak head normal form (`typename Reduce<left>::type`) after which the left side is in weak head normal form, so the next redex is this application:

```
typename
  Reduce<left>::type::template apply<right>::type
```

Finally the resulting expression is reduced as well.

1.4 EVALUATION

We solved the same problem with a hand-written C++ Template Metaprogram and with embedded lambda expressions. The task was producing a list of warnings containing the primes in an interval. We used a simple O(n^2) algorithm: for each `n` number in the interval the program determined individually whether it's a prime or not by testing whether numbers `[2..n/2]` are dividers of `n` or not.

1.4.1 Code size

We have compared the length of the code to write (debug and maintain) by counting the effective lines of code. Out native implementation was 34 lines long while the solution using embedded lambda expressions was only 14 lines long. It means that using embedded lambda expressions reduces the length of the code - in our experiment the difference was significant (lambda expressions were less than half as long as native template metaprograms).

1.4.2 Template depth

We have compared how deep template depth the two solutions require. The embedded lambda expressions exceeded the (default) maximum template depth of the GNU C++ compiler for inputs longer than 57 elements while the native implementation exceeded the maximum only for inputs longer than 109 elements. As we can see the limit of the embedded lambda expressions is lower (because our lambda expression implementation uses currying).

1.4.3 Compilation time

We have measured compilation time of generated metaprograms and compared it with a native implementation (using the same algorithm). We run the tests on a Linux PC with 1 GB memory and a 2.6 GHz Celeron CPU. We used the 4.2.4 version of the GNU C++ compiler with default options. We used the `time` command to measure compilation time and used the `user` part of it's output.

Figure 1.1 shows the compilation times (the horizontal axis is the length of the interval (the first element of the interval was always 2), the vertical axis shows the seconds spent on compilation.

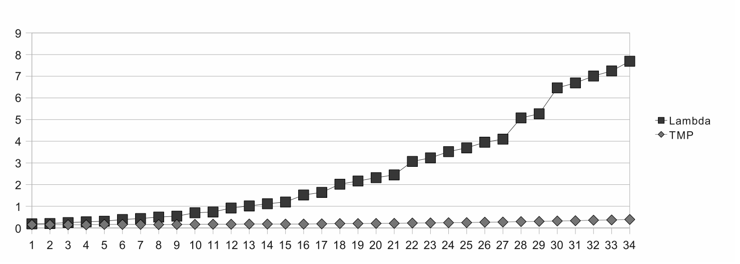

FIGURE 1.1. Compilation time

1.4.4 Debugging

Our solution doesn't make debugging metaprograms easier, the error messages are about the C++ templates representing the lambda expressions. But having a translation of lambda expressions to template metaprograms gives opportunities of extending the translated code with information about the original lambda expression making error messages more descriptive for developers.

1.5 RELATED WORK

1.5.1 FC++

FC++ is a C++ library providing runtime support for functional programming [20]. Using the tools the library provides functional programs can be written in C++ from which the expression graph is built and evaluated at runtime. They don't require any external tool (such as a translator) they use standard language features only. The library focuses on runtime execution.

1.5.2 Boost metaprogramming library

Boost has a template metaprogramming library called `boost::mpl` which implements several data types and algorithms following the logic of STL [11]. Our solution is designed to be compatible with it (the lambda expressions produced by our compiler are designed to be template metafunction classes taking one argument).

`Boost::mpl` has lambda expression support: the library provides tools to create lambda abstractions easily: placeholders (`_1`, `_2`, etc.) are provided and arguments of metafunctions can be replaced by them. The result of evaluating a metafunction with one (or more) placeholder argument is not directly usable, a metafunction called `lambda` generates a metafunction class from them. Using these lambda abstractions partial function applications can be implemented, but since `lambda` bounds every placeholder lambda abstractions with other lambda abstractions as their value can't be defined. For example λx.λy.+xy can't be expressed (and neither can be the `Y` fixpoint operator).

1.5.3 Boost lambda library

Boost has a library for implementing lambda abstractions in C++ [29]. It's main motivation is simplifying the creation of function objects for generic algorithms (such as STL algorithms). With the library function objects can be built from expressions (using placeholders). The lambda abstractions built using this library can be used at runtime.

1.5.4 EClean

Our solution is not the first attempt to express template metaprograms using a functional language. A Clean to Template Metaprogram translator has been written [22]. It uses a subset of Clean (EClean) as the source language which it creates template metaprograms from.

1.6 CONCLUSION AND FUTURE WORKS

Template metaprograms are difficult to write, debug and maintain. They can be tought of as purely functional programs [15]. We examined if they could be expressed in well known, easy to use functional languages.

Lambda expressions are well studied, easy to process structures capable of expressing everything higher level functional languages can [6]. We have examined how lambda expressions could be used to write template metaprograms in and provided a tool for compiling in-line lambda expressions into template metaprograms. Our solution demonstrates that any functional language (such as Haskell) can be compiled into template metaprograms.

We have seen that using lambda expressions simplify template metaprograms, make them easier to write and maintain. The C++ language was not designed for template metaprograms, they have to contain unnecessary syntactical elements. The use of lambda expressions eliminiates these unnecessary syntax elements and let the developer (or the reader of the code) focus on the functionality of the metaprogram. The use of higher level functional languages could simplify codes even further.

REFERENCES

[1] D. Abrahams, A. Gurtovoy, *C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, Boston, 2004.

[2] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001.

[3] ANSI/ISO C++ Committee, *Programming Languages – C++*, ISO/IEC 14882:1998(E), American National Standards Institute, 1998.

[4] T. H. Brus, C. J. D. van Eekelen, M. O. van Leer, M. J. Plasmeijer, *CLEAN: A language for functional graph rewriting*, Proc. of a conference on Functional programming languages and computer architecture, Springer-Verlag, 1987, pp.364-384.

[5] Zoltán Csörnyei and Gergely Dévai, *An introduction to the lambda-calculus*, Lecture Notes in Computer Science, Springer-Verlag, LNCS Vol. 5161, pp. 87-111 ISSN 0302-9743, ISBN 3-540-88058-5

[6] Zoltán Csörnyei, *Lambda kalkulus – A funkcionális programozá alapjai* Typotex, 2007, Budapest, ISBN: 978-963-9664-46-3

[7] Olaf Chitil, Zoltán Horváth, Viktória Zsók (Eds.): *Implementation and Application of Functional Languages*, Springer, 2008, [273], ISBN: 978-3-540-85372-5

[8] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, T. L. Veldhuizen, *Generative Programming and Active Libraries*, Springer-Verlag, 2000.

[9] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, 2000.

[10] Zoltán Horváth, Rinus Plasmeijer, Anna Soós, Viktória Zsók (Eds.): *Central European Functional Programming School*, Springer, 2008, [301], ISBN: 978-3-540-88058-5

[11] B. Karlsson, *Beyond the C++ Standard Library, An Introduction to Boost*, Addison-Wesley, 2005.

[12] P. Koopman, R. Plasmeijer, M. van Eeekelen, S. Smetsers, *Functional programming in Clean*, 2002

[13] D. R. Musser, A. A. Stepanov, *Algorithm-oriented Generic Libraries*, Software-practice and experience 27(7), 1994, pp.623-642.

[14] R. Plasmeijer, M. van Eeekelen, *Clean Language Report*, 2001.

[15] Zoltán Porkoláb, József Mihalicza, Ádám Sipos, *Debugging C++ template metaprograms,* In: Stan Jarzabek, Douglas C. Schmidt, Todd L. Veldhuizen (Eds.): Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings. ACM 2006, ISBN 1-59593-237-2, pp. 255-264.

[16] J. Siek, A. Lumsdaine, *Essential Language Support for Generic Programming*, Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, USA, pp 73-84.

[17] J. Siek, *A Language for Generic Programming,* PhD thesis, Indiana University, 2005.

[18] B. Stroustrup, *The C++ Programming Language Special Edition*, Addison-Wesley, 2000.

[19] G. Dos Reis, B. Stroustrup, *Specifying C++ concepts*, Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2006, pp. 295-308.

[20] B. McNamara, Y. Smaragdakis, *Functional programming in C++*, Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, pp.118-129, 2000.

[21] Simon L. Peyton Jones: *The Implementation of Functional Languages*, Prentice Hall, 1987, [445], ISBN: 0-13-453333-9 Pbk

[22] Ádám Sipos, Zoltán Porkoláb, Viktória Zsók: *Meta<fun> – Towards a functional-style interface for C++ template metaprograms* In Frentiu et al ed.: Studia Universitatis Babes-Bolyai Informatica LIII, 2008/2, Cluj-Napoca, 2008, pp. 55-66.

[23] E. Unruh, *Prime number computation,* ANSI X3J16-94-0075/ISO WG21-462.

[24] D. Vandevoorde, N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley, 2003.

[25] T. Veldhuizen, *Using C++ Template Metaprograms,* C++ Report vol. 7, no. 4, 1995, pp. 36-43.

[26] T. Veldhuizen, *Expression Templates*, C++ Report vol. 7, no. 5, 1995, pp. 26-31.

[27] T. Veldhuizen, *C++ Templates are Turing Complete*

[28] I. Zólyomi, Z. Porkoláb, *Towards a template introspection library*, LNCS Vol.3286 (2004), pp.266-282.

[29] Boost Libraries.
http://www.boost.org/