

---

# Phoenix 2.0

Joel de Guzman

Dan Marsden

Copyright © 2002-2005 Joel de Guzman, Dan Marsden

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE\_1\_0.txt or copy at [http://www.boost.org/LICENSE\\_1\\_0.txt](http://www.boost.org/LICENSE_1_0.txt))

## Table of Contents

Preface .....	2
Introduction .....	3
Starter Kit .....	4
Values .....	4
References .....	5
Arguments .....	5
Composites .....	6
Lazy Operators .....	6
Lazy Statements .....	7
Construct, New, Delete, Casts .....	8
Lazy Functions .....	8
More .....	9
Basics .....	9
Organization .....	12
Actors .....	14
Primitives .....	15
Arguments .....	15
Values .....	17
References .....	18
Constant References .....	18
Nothing .....	18
Composite .....	19
Function .....	19
Operator .....	20
Statement .....	23
Object .....	29
Scope .....	31
Bind .....	36
Container .....	38
Algorithm .....	41
Inside Phoenix .....	44
Actors In Detail .....	44
Actor Example .....	48
Composites In Detail .....	48
Composing .....	50
Extending .....	53
Wrap Up .....	53
Acknowledgement .....	54
References .....	54

## Preface

*Functional programming is so called because a program consists entirely of functions. The main program itself is written as a function which receives the program's input as its argument and delivers the program's output as its result. Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions until at the bottom level the functions are language primitives.*

**John Hughes**-- *Why Functional Programming Matters*



## Description

Phoenix enables Functional Programming (FP) in C++. The design and implementation of Phoenix is highly influenced by [FC++](#) by Yannis Smaragdakis and Brian McNamara and the [BLL](#) (Boost Lambda Library) by Jaakko Jaarvi and Gary Powell. Phoenix is a blend of FC++ and BLL using the implementation techniques used in the [Spirit](#) inline parser. Phoenix version 2, this version, will probably be the last release of the library. Phoenix v2 will be the basis of the Phoenix and [BLL](#) merger.

Phoenix is a header only library. It is extremely modular by design. One can extract and use only a small subset of the full library, literally tearing the library into small pieces, without fear that the pieces won't work anymore. The library is organized in highly independent modules and layers.

## How to use this manual

The Phoenix library is organized in logical modules. This documentation provides a user's guide and reference for each module in the library. A simple and clear code example is worth a hundred lines of documentation; therefore, the user's guide is presented with abundant examples annotated and explained in step-wise manner. The user's guide is based on examples: lots of them.

As much as possible, forward information (i.e. citing a specific piece of information that has not yet been discussed) is avoided in the user's manual portion of each module. In many cases, though, it is unavoidable that advanced but related topics not be interspersed with the normal flow of discussion. To alleviate this problem, topics categorized as "advanced" may be skipped at first reading.

Some icons are used to mark certain topics indicative of their relevance. These icons precede some text to indicate:

**Table 1. Icons**

Icon	Name	Meaning
	Note	Information provided is auxiliary but will give the reader a deeper insight into a specific topic. May be skipped.
	Alert	Information provided is of utmost importance.
	Tip	A potentially useful and helpful piece of information.

This documentation is automatically generated by Spirit QuickBook documentation tool. QuickBook can be found in the [Spirit Repository](#).

## Support

Please direct all questions to Spirit's mailing list. You can subscribe to the [Spirit Mailing List](#). The mailing list has a searchable archive. A search link to this archive is provided in [Spirit](#)'s home page. You may also read and post messages to the mailing list through [Spirit General NNTP news portal](#) (thanks to [Gmane](#)). The news group mirrors the mailing list. Here is a link to the archives: <http://news.gmane.org/gmane.comp.parsers.spirit.general>.

## ...To my dear daughter, Phoenix

## Introduction



The Phoenix library enables FP techniques such as higher order functions, *lambda* (unnamed functions), *currying* (partial function application) and lazy evaluation in C++. The focus is more on usefulness and practicality than purity, elegance and strict adherence to FP principles.

FP is a programming discipline that is not at all tied to a specific language. FP as a programming discipline can, in fact, be applied to many programming languages. In the realm of C++ for instance, we are seeing more FP techniques being applied. C++ is sufficiently rich to support at least some of the most important facets of FP. C++ is a multiparadigm programming language. It is not only procedural. It is not only object oriented. Beneath the core of the standard C++ library, a closer look into STL gives us a glimpse of FP already in place. It is obvious that the authors of STL know and practice FP. In the near future, we shall surely see more FP trickle down into the mainstream.

The truth is, most of the FP techniques can coexist quite well with the standard object oriented and imperative programming paradigms. When we are using STL algorithms and functors (function objects) for example, we are already doing FP. Phoenix is an evolutionary next step.

## Starter Kit

Most "quick starts" only get you a few blocks from where you are. From there, you are on your own. Yet, typically, you'd want to get to the next city. This starter kit shall be as minimal as possible, yet packed as much power as possible.

So you are busy and always on the go. You do not wish to spend a lot of time studying the library. You wish to be spared the details for later when you need it. For now, all you need to do is to get up to speed as quickly as possible and start using the library. If this is the case, this is the right place to start.

This chapter is by no means a thorough discourse of the library. For more information on Phoenix, please take some time to read the rest of the User's Guide. Yet, if you just want to use the library quickly, now, this chapter will probably suffice. Rather than taking you to the details of the library, we shall try to provide you with annotated exemplars instead. Hopefully, this will get you into high gear quickly.

## Functors everywhere

Phoenix is built on function objects (functors). The functor is the main building block. We compose functors to build more complex functors... to build more complex functors... and so on. Almost everything is a functor.



### Note

Functors are so ubiquitous in Phoenix that, in the manual, the words "*functor*" and "*function*" are used interchangeably.

## Values

Values are functions! Examples:

```
val(3)
val("Hello, World")
```

The first evaluates to a nullary function (a function taking no arguments) that returns an `int`, 3. The second evaluates to a nullary function that returns a `char const&[13]`, "Hello, World".

## Lazy Evaluation

Confused? `val(3)` is a unary function, you say? Yes it is. However, read carefully: "*evaluates to a nullary function*". `val(3)` evaluates to (returns) a nullary function. Aha! `val(3)` returns a function! So, since `val(3)` returns a function, you can invoke it. Example:

```
cout << val(3)() << endl;
```

(See [values.cpp](#))



Learn more about values [here](#).

The second function call (the one with no arguments) calls the nullary function which then returns 3. The need for a second function call is the reason why the function is said to be **Lazily Evaluated**. The first call doesn't do anything. You need a second call to finally evaluate the thing. The first call lazily evaluates the function; i.e. doesn't do anything and defers the evaluation for later.

## Callbacks

It may not be immediately apparent how lazy evaluation can be useful by just looking at the example above. Putting the first and second function call in a single line is really not very useful. However, thinking of `val(3)` as a callback function (and in most cases they are actually used that way), will make it clear. Example:

```
template <typename F>
void print(F f)
{
    cout << f() << endl;
}

int
main()
{
    print(val(3));
    print(val("Hello World"));
    return 0;
}
```

(See [callback.cpp](#))

## References

References are functions. They hold a reference to a value stored somewhere. For example, given:

```
int i = 3;
char const* s = "Hello World";
```

we create references to `i` and `s` this way:

```
ref(i)
ref(s)
```

Like `val`, the expressions above evaluates to a nullary function; the first one returning an `int&`, and the second one returning a `char const*&`.

(See [references.cpp](#))



Learn more about references [here](#).

## Arguments

Arguments are also functions? You bet!

Until now, we have been dealing with expressions returning a nullary function. Arguments, on the other hand, evaluate to an N-ary function. An argument represents the Nth argument. There are a few predefined arguments `arg1`, `arg2`, `arg3`, `arg4` and so on (and it's **BLL** counterparts: `_1`, `_2`, `_3`, `_4` and so on). Examples:

```
arg1 // one-or-more argument function that returns its first argument
arg2 // two-or-more argument function that returns its second argument
arg3 // three-or-more argument function that returns its third argument
```

`argN` returns the Nth argument. Examples:

```
int i = 3;
char const* s = "Hello World";
cout << arg1(i) << endl;           // prints 3
cout << arg2(i, s) << endl;         // prints "Hello World"
```

(See [arguments.cpp](#))



Learn more about arguments [here](#).

## Composites

What we have seen so far, are what are called **primitives**. You can think of primitives (such as values, references and arguments) as atoms.

Things start to get interesting when we start *composing* primitives to form **composites**. The composites can, in turn, be composed to form even more complex composites.

## Lazy Operators

You can use the usual set of operators to form composites. Examples:

```
arg1 * arg1
ref(x) = arg1 + ref(z)
arg1 = arg2 + (3 * arg3)
ref(x) = arg1[arg2] // assuming arg1 is indexable and arg2 is a valid index
```

Note the expression: `3 * arg3`. This expression is actually a short-hand equivalent to: `val(3) * arg3`. In most cases, like above, you can get away with it. But in some cases, you will have to explicitly wrap your values in `val`. Rules of thumb:

- In a binary expression (e.g. `3 * arg3`), at least one of the operands must be a phoenix primitive or composite.
- In a unary expression (e.g. `arg1++`), the single operand must be a phoenix primitive or composite.

If these basic rules are not followed, the result is either in error, or is immediately evaluated. Some examples:

```

ref(x) = 123      // lazy
x = 123          // immediate

ref(x)[0]         // lazy
x[0]             // immediate

ref(x)[ref(i)]   // lazy
ref(x)[i]         // lazy (equivalent to ref(x)[val(i)])
x[ref(i)]        // illegal (x is not a phoenix primitive or composite)
ref(x[ref(i)])   // illegal (x is not a phoenix primitive or composite)

```



Learn more about operators [here](#).

## First Practical Example

We've covered enough ground to present a real world example. We want to find the first odd number in an STL container. Normally we use a functor (function object) or a function pointer and pass that in to STL's `find_if` generic function:

Write a function:

```

bool
is_odd(int arg1)
{
    return arg1 % 2 == 1;
}

```

Pass a pointer to the function to STL's `find_if` algorithm:

```
find_if(c.begin(), c.end(), &is_odd)
```

Using Phoenix, the same can be achieved directly with a one-liner:

```
find_if(c.begin(), c.end(), arg1 % 2 == 1)
```

The expression `arg1 % 2 == 1` automagically creates a functor with the expected behavior. In FP, this unnamed function is called a lambda function. Unlike the function pointer version, which is monomorphic (expects and works only with a fixed type `int` argument), the Phoenix version is fully polymorphic and works with any container (of ints, of longs, of bignum, etc.) as long as its elements can handle the `arg1 % 2 == 1` expression.

(See [find\\_if.cpp](#))



...That's it, we're done. Well if you wish to know a little bit more, read on...

## Lazy Statements

Lazy statements? Sure. There are lazy versions of the C++ statements we all know and love. For example:

```
if_(arg1 > 5)
    cout << arg1
```

Say, for example, we wish to print all the elements that are greater than 5 (separated by a comma) in a vector. Here's how we write it:

```
for_each(v.begin(), v.end(),
    if_(arg1 > 5)
    [
        cout << arg1 << ", "
    ]
);
```

(See [if.cpp](#))



Learn more about statements [here](#).

## Construct, New, Delete, Casts

You'll probably want to work with objects. There are lazy versions of constructor calls, new, delete and the suite of C++ casts. Examples:

```
construct<std::string>(arg1, arg2) // constructs a std::string from arg1, arg2
new_<std::string>(arg1, arg2)      // makes a new std::string from arg1, arg2
delete_(arg1)                      // deletes arg1 (assumed to be a pointer)
static_cast_<int*>(arg1)          // static_cast's arg1 to an int*
```



### Note

Take note that, by convention, names that conflict with C++ reserved words are appended with a single trailing underscore '\_'



Learn more about this [here](#).

## Lazy Functions

As you write more lambda functions, you'll notice certain patterns that you wish to refactor as reusable functions. When you reach that point, you'll wish that ordinary functions can co-exist with phoenix functions. Unfortunately, the *immediate* nature of plain C++ functions make them incompatible.

Lazy functions are your friends. The library provides a facility to make lazy functions. The code below is a rewrite of the `is_odd` function using the facility:

```

struct is_odd_Impl
{
    template <typename Arg>
    struct result
    {
        typedef bool type;
    };

    template <typename Arg>
    bool operator()(Arg arg1) const
    {
        return arg1 % 2 == 1;
    }
};

function<is_odd_Impl> is_odd;

```

## Things to note:

- `result` is a nested metafunction that reflects the return type of the function (in this case, `bool`). This makes the function fully polymorphic: It can work with arbitrary `Arg` types.
- There are as many `Args` in the `result` metafunction as in the actual `operator()`.
- `is_odd_Impl` implements the function.
- `is_odd`, an instance of `function<is_odd_Impl>`, is the lazy function.

Now, `is_odd` is a truly lazy function that we can use in conjunction with the rest of phoenix. Example:

```
find_if(c.begin(), c.end(), is_odd(arg1));
```

(See [function.cpp](#))

## Predefined Lazy Functions

The library is chock full of STL savvy, predefined lazy functions covering the whole of the STL containers, iterators and algorithms. For example, there are lazy versions of container related operations such as `assign`, `at`, `back`, `begin`, `pop_back`, `pop_front`, `push_back`, `push_front`, etc. (See [Container](#)).

## More

As mentioned earlier, this chapter is not a thorough discourse of the library. It is meant only to cover enough ground to get you into high gear as quickly as possible. Some advanced stuff is not discussed here (e.g. [Scopes](#)); nor are features that provide alternative (short-hand) ways to do the same things (e.g. [Bind](#) vs. Lazy Functions).



...If you still wish to learn more, the read on...

## Basics

Almost everything is a function in the Phoenix library that can be evaluated as `f(a1, a2, ..., an)`, where  $n$  is the function's arity, or number of arguments that the function expects. Operators are also functions. For example, `a + b` is just a function with arity == 2 (or binary). `a + b` is the same as `add(a, b)`, `a + b + c` is the same as `add(add(a, b), c)`.



### Note

Amusingly, functions may even return functions. We shall see what this means in a short while.

## Partial Function Application

Think of a function as a black box. You pass arguments and it returns something back. The figure below depicts the typical scenario.



A fully evaluated function is one in which all the arguments are given. All functions in plain C++ are fully evaluated. When you call the `sin(x)` function, you have to pass a number `x`. The function will return a result in return: the sin of `x`. When you call the `add(x, y)` function, you have to pass two numbers `x` and `y`. The function will return the sum of the two numbers. The figure below is a fully evaluated `add` function.



A partially applied function, on the other hand, is one in which not all the arguments are supplied. If we are able to partially apply the function `add` above, we may pass only the first argument. In doing so, the function does not have all the required information it needs to perform its task to compute and return a result. What it returns instead is another function, a lambda function --another black box. Unlike the original `add` function which has an arity of 2, the resulting lambda function has an arity of 1. Why? because we already supplied part of the input: 2



Now, when we shove in a number into our lambda function, it will return 2 plus whatever we pass in. The lambda function essentially remembers 1) the original function, `add`, and 2) the partial input, 2. The figure below illustrates a case where we pass 3 to our lambda function, which then returns 5:



Obviously, partially applying the `add` function, as we see above, cannot be done directly in C++ where we are expected to supply all the arguments that a function expects. That's where the Phoenix library comes in. The library provides the facilities to do partial function application.

## STL and higher order functions

So, what's all the fuss? What makes partial function application so useful? Recall our original example in the [previous section](#):

```
find_if(c.begin(), c.end(), arg1 % 2 == 1)
```

The expression `arg1 % 2 == 1` evaluates to a lambda function. `arg1` is a placeholder for an argument to be supplied later. Hence, since there's only one unsupplied argument, the lambda function has an arity 1. It just so happens that `find_if` supplies the unsupplied argument as it loops from `c.begin()` to `c.end()`.



## Note

Higher order functions are functions which can take other functions as arguments, and may also return functions as results. Higher order functions are functions that are treated like any other objects and can be used as arguments and return values from functions.

## Lazy Evaluation

In Phoenix, to put it more accurately, function evaluation has two stages:

1. Partial application
2. Final evaluation

The first stage is handled by a set of generator functions. These are your front ends (in the client's perspective). These generators create (through partial function application), higher order functions that can be passed on just like any other function pointer or function object. The second stage, the actual function call, can be invoked or executed anytime in the future, or not at all; hence "*lazy*".

If we look more closely, the first step involves partial function application:

```
arg1 % 2 == 1
```

The second step is the actual function invocation (done inside the `find_if` function). These are the back-ends (often, the final invocation is never actually seen by the client). In our example, the `find_if`, if we take a look inside, we'll see something like:

```
template <class InputIterator, class Predicate>
InputIterator
find_if(InputIterator first, InputIterator last, Predicate pred)
{
    while (first != last && !pred(*first)) // <--- The lambda function is called here
        ++first;                         //      passing in *first
    return first;
}
```

Again, typically, we, as clients, see only the first step. However, in this document and in the examples and tests provided, don't be surprised to see the first and second steps juxtaposed in order to illustrate the complete semantics of Phoenix expressions. Examples:

```
int x = 1;
int y = 2;

cout << (arg1 % 2 == 1)(x) << endl; // prints 1 or true
cout << (arg1 % 2 == 1)(y) << endl; // prints 0 or false
```

## Forwarding Function Problem

Usually, we, as clients, write the call-back functions while libraries (such as STL) provide the callee (e.g. `find_if`). In case the role is reversed, e.g. if you have to write an STL algorithm that takes in a predicate, or develop a GUI library that accepts event handlers, you have to be aware of a little known problem in C++ called the "[Forwarding Function Problem](#)".

Look again at the code above:

```
(arg1 % 2 == 1)(x)
```

Notice that, in the second-stage (the final evaluation), we used a variable `x`. Be aware that the second stage cannot accept non-const temporaries and literal constants. Hence, this will fail:

```
(arg1 % 2 == 1)(123) // Error!
```

Disallowing non-const rvalues partially solves the "Forwarding Function Problem" but prohibits code like above.

## Polymorphic Functions

Unless otherwise noted, Phoenix generated functions are fully polymorphic. For instance, the `add` example above can apply to integers, floating points, user defined complex numbers or even strings. Example:

```
std::string h("Hello");
char const* w = "World";
std::string r = add(arg1, arg2)(h, w);
```

evaluates to `std::string("Hello World")`. The observant reader might notice that this function call in fact takes in heterogeneous arguments where `arg1` is of type `std::string` and `arg2` is of type `char const*`. `add` still works because the C++ standard library allows the expression `a + b` where `a` is a `std::string` and `b` is a `char const*`.

## Organization

Care and attention to detail was given, painstakingly, to the design and implementation of Phoenix.

The library is organized in four layers:

Intrinsic			Algorithm		
Function	Operator	Statement	Object	Scope	Bind
Primitives			Composite		
Actor					

The modules are orthogonal, with no cyclic dependencies. Lower layers do not depend on higher layers. Modules in a layer do not depend on other modules in the same layer. This means, for example, that Bind can be completely discarded if it is not required; or one could perhaps take out Operator and Statement and just use Function, which may be desireable in a pure FP application.

The library has grown from the original Phoenix but still comprises only header files. There are no object files to link against.

## Core

The lowest two layers comprise the core.

The Actor is the main concept behind the library. Lazy functions are abstracted as actors. There are only 2 kinds of actors:

1. Primitives

## 2. Composites

Primitives provide the basic building blocks of functionality within Phoenix. Composites are used to combine these primitives together to provide more powerful functionality.

Composites are composed of zero or more actors. Each actor in a composite can again be another composite.

**Table 2. Modules**

Module	Description
Function	Lazy functions support (e.g. add)
Operator	Lazy operators support (e.g. +)
Statement	Lazy statements (e.g. if_, while_)
Object	Lazy casts (e.g. static_cast_), object creation destruction (e.g. new_, delete_)
Scope	Support for scopes, local variables and lambda-lambda
Bind	Lazy functions from free functions, member functions or member variables.
Container	Set of predefined "lazy" functions that work on STL containers and sequences (e.g. push_back).
Algorithm	Set of predefined "lazy" versions of the STL algorithms (e.g. find_if).

Each module is defined in a header file with the same name. For example, the core module is defined in `<boost/spirit/home/phoenix/core.hpp>`.

**Table 3. Includes**

<b>Module</b>	<b>File</b>
Core	#include <boost/spirit/home/phoenix/core.hpp>
Function	#include <boost/spirit/home/phoenix/function.hpp>
Operator	#include <boost/spirit/home/phoenix/operator.hpp>
Statement	#include <boost/spirit/home/phoenix/statement.hpp>
Object	#include <boost/spirit/home/phoenix/object.hpp>
Scope	#include <boost/spirit/home/phoenix scope.hpp>
Bind	#include <boost/spirit/home/phoenix/bind.hpp>
Container	#include <boost/spirit/home/phoenix/container.hpp>
Algorithm	#include <boost/spirit/home/phoenix/algorithm.hpp>



Finer grained include files are available per feature; see the succeeding sections.

## Actors

The Actor is the main concept behind the library. Actors are function objects. An actor can accept 0 to `PHOENIX_LIMIT` arguments.



### Note

You can set `PHOENIX_LIMIT`, the predefined maximum arity an actor can take. By default, `PHOENIX_LIMIT` is set to 10.

Phoenix supplies an `actor` class template whose specializations model the Actor concept. `actor` has one template parameter, `Eval`, that supplies the smarts to evaluate the resulting function.

```

template <typename Eval>
struct actor : Eval
{
    return_type
    operator()() const;

    template <typename T0>
    return_type
    operator()(T0& _0) const;

    template <typename T0, typename T1>
    return_type
    operator()(T0& _0, T1& _1) const;

    //...
};

```

The actor class accepts the arguments through a set of function call operators for 0 to PHOENIX\_LIMIT arities (Don't worry about the details, for now. Note, for example, that we skimp over the details regarding `return_type`). The arguments are then forwarded to the actor's `Eval` for evaluation.

## Primitives

Actors are composed to create more complex actors in a tree-like hierarchy. The primitives are atomic entities that are like the leaves in the tree. Phoenix is extensible. New primitives can be added anytime. Right out of the box, there are only a few primitives. This section shall deal with these preset primitives.

## Arguments

```
#include <boost/spirit/home/phoenix/core/argument.hpp>
```

We use an instance of:

```
actor<argument<N> >
```

to represent the Nth function argument. The argument placeholder acts as an imaginary data-bin where a function argument will be placed.

## Predefined Arguments

There are a few predefined instances of `actor<argument<N> >` named `arg1..argN`, and its **BLL** counterpart `_1.._N`. (where N is a predefined maximum).

Here are some sample preset definitions of `arg1..argN`

```
actor<argument<0> > const arg1 = argument<0>();
actor<argument<1> > const arg2 = argument<1>();
actor<argument<2> > const arg3 = argument<2>();
```

and its **BLL** `_1.._N` style counterparts:

```
actor<argument<0> > const _1 = argument<0>();
actor<argument<1> > const _2 = argument<1>();
actor<argument<2> > const _3 = argument<2>();
```



## Note

You can set `PHOENIX_ARG_LIMIT`, the predefined maximum placeholder index. By default, `PHOENIX_ARG_LIMIT` is set to `PHOENIX_LIMIT` (See [Actors](#)).

## User Defined Arguments

When appropriate, you can define your own `argument<N>` names. For example:

```
actor<argument<0> > x; // note zero based index
```

`x` may now be used as a parameter to a lazy function:

```
add(x, 6)
```

which is equivalent to:

```
add(arg1, 6)
```

## Evaluating an Argument

An argument, when evaluated, selects the Nth argument from the those passed in by the client.

For example:

```
char      c = 'A';
int      i = 123;
const char* s = "Hello World";

cout << arg1(c) << endl;           // Get the 1st argument: c
cout << arg1(i, s) << endl;         // Get the 1st argument: i
cout << arg2(i, s) << endl;         // Get the 2nd argument: s
```

will print out:

```
A
123
Hello World
```

## Extra Arguments

In C and C++, a function can have extra arguments that are not at all used by the function body itself. These extra arguments are simply ignored.

Phoenix also allows extra arguments to be passed. For example, recall our original `add` function:

```
add(arg1, arg2)
```

We know now that partially applying this function results to a function that expects 2 arguments. However, the library is a bit more lenient and allows the caller to supply more arguments than is actually required. Thus, `add` actually allows *2 or more* arguments. For instance, with:

```
add(arg1, arg2)(x, y, z)
```

the third argument `z` is ignored. Taking this further, in-between arguments are also ignored. Example:

```
add(arg1, arg5)(a, b, c, d, e)
```

Here, arguments `b`, `c`, and `d` are ignored. The function `add` takes in the first argument (`arg1`) and the fifth argument (`arg5`).



## Note

There are a few reasons why enforcing strict arity is not desireable. A case in point is the callback function. Typical callback functions provide more information than is actually needed. Lambda functions are often used as callbacks.

## Values

```
#include <boost/spirit/home/phoenix/core/value.hpp>
```

Whenever we see a constant in a partially applied function, an

```
actor<value<T> >
```

(where `T` is the type of the constant) is automatically created for us. For instance:

```
add(arg1, 6)
```

Passing a second argument, `6`, an `actor<value<int> >` is implicitly created behind the scenes. This is also equivalent to:

```
add(arg1, val(6))
```

`val(x)` generates an `actor<value<T> >` where `T` is the type of `x`. In most cases, there's no need to explicitly use `val`, but, as we'll see later on, there are situations where this is unavoidable.

## Evaluating a Value

Like arguments, values are also actors. As such, values can be evaluated. Invoking a value gives the value's identity. Example:

```
cout << val(3)() << val("Hello World")();
```

prints out "3 Hello World".

## References

```
#include <boost/spirit/home/phoenix/core/reference.hpp>
```

Values are immutable constants. Attempting to modify a value will result in a compile time error. When we want the function to modify the parameter, we use a reference instead. For instance, imagine a lazy function `add_assign`:

```
void add_assign(T& x, T y) { x += y; } // pseudo code
```

Here, we want the first function argument, `x`, to be mutable. Obviously, we cannot write:

```
add_assign(1, 2) // error first argument is immutable
```

In C++, we can pass in a reference to a variable as the first argument in our example above. Yet, by default, the library forces arguments passed to partially applied functions functions to be immutable values (see [Values](#)). To achieve our intent, we use:

```
actor<reference<T> >
```

This is similar to `actor<value<T> >` above but instead holds a reference to a variable.

We normally don't instantiate `actor<reference<T> >` objects directly. Instead we use `ref`. For example (where `i` is an `int` variable):

```
add_assign(ref(i), 2)
```

## Evaluating a Reference

References are actors. Hence, references can be evaluated. Such invocation gives the references's identity. Example:

```
int i = 3;
char const* s = "Hello World";
cout << ref(i)() << ref(s)();
```

prints out "3 Hello World"

## Constant References

```
#include <boost/spirit/home/phoenix/core/reference.hpp>
```

Another free function  `cref(cv)` may also be used.  `cref(cv)` creates an `actor<reference<T const&> >` object. This is similar to `actor<value<T> >` but when the data to be passed as argument to a function is heavy and expensive to copy by value, the  `cref(cv)` offers a lighter alternative.

## Nothing

```
#include <boost/spirit/home/phoenix/core/nothing.hpp>
```

Finally, the `actor<null_actor>` does nothing; (a "bum", if you will :-). There's a sole `actor<null_actor>` instance named "nothing". This actor is actually useful in situations where we don't want to do anything. (See [for\\_ Statement](#) for example).

# Composite

Actors may be combined in a multitude of ways to form composites. Composites are actors that are composed of zero or more actors. Composition is hierarchical. An element of the composite can be a primitive or again another composite. The flexibility to arbitrarily compose hierarchical structures allows us to form intricate constructions that model complex functions, statements and expressions.

A composite is-a tuple of 0..N actors. N is the predefined maximum actors a composite can take.



## Note

You can set `PHOENIX_COMPOSITE_LIMIT`, the predefined maximum actors a composite can take. By default, `PHOENIX_COMPOSITE_LIMIT` is set to `PHOENIX_LIMIT` (See [Actors](#)).

As mentioned, each of the actors A0..AN can, in turn, be another composite, since a composite is itself an actor. This makes the composite a recursive structure. The actual evaluation is handled by a composite specific eval policy.

# Function

```
#include <boost/spirit/home/phoenix/function/function.hpp>
```

The `function` class template provides a mechanism for implementing lazily evaluated functions. Syntactically, a lazy function looks like an ordinary C/C++ function. The function call looks familiar and feels the same as ordinary C++ functions. However, unlike ordinary functions, the actual function execution is deferred.

Unlike ordinary function pointers or functor objects that need to be explicitly bound through the `bind` function (see [Bind](#)), the argument types of these functions are automatically lazily bound.

In order to create a lazy function, we need to implement a model of the `FunctionEval` concept. For a function that takes N arguments, a model of `FunctionEval` must provide:

- An `operator()` that implements that takes N arguments, and implements the function logic.
- A nested metafunction `result<A1, ... AN>` that takes the types of the N arguments to the function and returns the result type of the function. (There is a special case for function objects that accept no arguments. Such nullary functors are only required to define a `typedef result_type` that reflects the return type of its `operator()`).

For example, the following type implements the `FunctionEval` concept, in order to provide a lazy factorial function:

```
struct factorial_impl
{
    template <typename Arg>
    struct result
    {
        typedef Arg type;
    };

    template <typename Arg>
    Arg operator()(Arg n) const
    {
        return (n <= 0) ? 1 : n * this->operator()(n-1);
    }
};
```

(See [factorial.cpp](#))

Having implemented the `factorial_impl` type, we can declare and instantiate a lazy `factorial` function this way:

```
function<factorial_impl> factorial;
```

Invoking a lazy function such as `factorial` does not immediately execute the function object `factorial_impl`. Instead, an [actor](#) object is created and returned to the caller. Example:

```
factorial(arg1)
```

does nothing more than return an actor. A second function call will invoke the actual factorial function. Example:

```
int i = 4;
cout << factorial(arg1)(i);
```

will print out "24".

Take note that in certain cases (e.g. for function objects with state), an instance of the model of `FunctionEval` may be passed on to the constructor. Example:

```
function<factorial_impl> factorial(ftor);
```

where `ftor` is an instance of `factorial_impl` (this is not necessary in this case as `factorial_impl` does not require any state).



Take care though when using function objects with state because they are often copied repeatedly, and state may change in one of the copies, rather than the original.

## Operator

This facility provides a mechanism for lazily evaluating operators. Syntactically, a lazy operator looks and feels like an ordinary C/C++ infix, prefix or postfix operator. The operator application looks the same. However, unlike ordinary operators, the actual operator execution is deferred. Samples:

```
arg1 + arg2
1 + arg1 * arg2
1 / -arg1
arg1 < 150
```

We have seen the lazy operators in action (see [Quick Start](#)). Let's go back and examine them a little bit further:

```
find_if(c.begin(), c.end(), arg1 % 2 == 1)
```

Through operator overloading, the expression `arg1 % 2 == 1` actually generates an actor. This actor object is passed on to STL's `find_if` function. From the viewpoint of STL, the composite is simply a function object expecting a single argument of the containers `value_type`. For each element in `c`, the element is passed on as an argument `arg1` to the actor (function object). The actor checks if this is an odd value based on the expression `arg1 % 2 == 1` where `arg1` is replaced by the container's element.

Like lazy functions (see [function](#)), lazy operators are not immediately executed when invoked. Instead, an actor (see [actors](#)) object is created and returned to the caller. Example:

```
(arg1 + arg2) * arg3
```

does nothing more than return an actor. A second function call will evaluate the actual operators. Example:

```
int i = 4, j = 5, k = 6;  
cout << ((arg1 + arg2) * arg3)(i, j, k);
```

will print out "54".

Operator expressions are lazily evaluated following four simple rules:

1. A binary operator, except  $\rightarrow^*$  will be lazily evaluated when *at least* one of its operands is an actor object (see [actors](#)).
2. Unary operators are lazily evaluated if their argument is an actor object.
3. Operator  $\rightarrow^*$  is lazily evaluated if the left hand argument is an actor object.
4. The result of a lazy operator is an actor object that can in turn allow the applications of rules 1 and 2.

For example, to check the following expression is lazily evaluated:

```
-(arg1 + 3 + 6)
```

1. Following rule 1,  $\text{arg1} + 3$  is lazily evaluated since  $\text{arg1}$  is an actor (see [primitives](#)).
2. The result of this  $\text{arg1} + 3$  expression is an actor object, following rule 4.
3. Continuing,  $\text{arg1} + 3 + 6$  is again lazily evaluated. Rule 2.
4. By rule 4 again, the result of  $\text{arg1} + 3 + 6$  is an actor object.
5. As  $\text{arg1} + 3 + 6$  is an actor,  $-(\text{arg1} + 3 + 6)$  is lazily evaluated. Rule 2.

Lazy-operator application is highly contagious. In most cases, a single  $\text{argN}$  actor infects all its immediate neighbors within a group (first level or parenthesized expression).

Note that at least one operand of any operator must be a valid actor for lazy evaluation to take effect. To force lazy evaluation of an ordinary expression, we can use `ref(x)`, `val(x)` or `cref(x)` to transform an operand into a valid actor object (see [primitives](#)). For example:

```
1 << 3;      // Immediately evaluated
val(1) << 3; // Lazily evaluated
```

## Supported operators

### Unary operators

```
prefix: ~, !, -, +, ++, --, & (reference), * (dereference)
postfix: ++, --
```

### Binary operators

```
=, [ ], +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
+, -, *, /, %, &, |, ^, <<, >>
==, !=, <, >, <=, >=
&&, ||, ->*
```

### Ternary operator

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

The ternary operator deserves special mention. Since C++ does not allow us to overload the conditional expression: `c ? a : b`, the `if_else` pseudo function is provided for this purpose. The behavior is identical, albeit in a lazy manner.

### Member pointer operator

```
a->*member_object_pointer
a->*member_function_pointer
```

The left hand side of the member pointer operator must be an actor returning a pointer type. The right hand side of the member pointer operator may be either a pointer to member object or pointer to member function.

If the right hand side is a member object pointer, the result is an actor which, when evaluated, returns a reference to that member. For example:

```
struct A
{
    int member;
};

A* a = new A;
...

(arg1->*&A::member)(a); // returns member a->member
```

If the right hand side is a member function pointer, the result is an actor which, when invoked, calls the specified member function. For example:

```

struct A
{
    int func(int);
};

A* a = new A;
int i = 0;

(arg1->*&A::func)(arg2)(a, i); // returns a->func(i)

```

**Table 4. Include Files**

<b>Operators</b>	<b>File</b>
-, +, ++, --, +=, -=, *=, /=, %=, *, /, %	#include <boost/spirit/home/phoenix/operator/arithmetic.hpp>
&=,  =, ^=, <<=, >>=, &,  , ^, <<, >>	#include <boost/spirit/home/phoenix/operator/bitwise.hpp>
==, !=, <, <=, >, >=	#include <boost/spirit/home/phoenix/operator/comparison.hpp>
<<, >>	#include <boost/spirit/home/phoenix/operator/io.hpp>
!, &&,	#include <boost/spirit/home/phoenix/operator/logical.hpp>
&x, *p, =, []	#include <boost/spirit/home/phoenix/operator/self.hpp>
if_else(c, a, b)	#include <boost/spirit/home/phoenix/operator/if_else.hpp>
->*	#include <boost/spirit/home/phoenix/operator/member.hpp>

## Statement

### Lazy statements...

The primitives and composite building blocks presented so far are sufficiently powerful to construct quite elaborate structures. We have presented lazy-functions and lazy-operators. How about lazy-statements? First, an appetizer:

Print all odd-numbered contents of an STL container using `std::for_each` ([all\\_odds.cpp](#)):

```

for_each(c.begin(), c.end(),
    if_(arg1 % 2 == 1)
    [
        cout << arg1 << ' '
    ]
);

```

Huh? Is that valid C++? Read on...

Yes, it is valid C++. The sample code above is as close as you can get to the syntax of C++. This stylized C++ syntax differs from actual C++ code. First, the `if` has a trailing underscore. Second, the block uses square brackets instead of the familiar curly braces {}.



## Note

### C++ in C++?

In as much as [Spirit](#) attempts to mimic EBNF in C++, Phoenix attempts to mimic C++ in C++!!!

Here are more examples with annotations. The code almost speaks for itself.

## Block Statement

```
#include <boost/spirit/home/phoenix/statement/sequence.hpp>
```

Syntax:

```
statement,  
statement,  
....  
statement
```

Basically, these are comma separated statements. Take note that unlike the C/C++ semicolon, the comma is a separator put **in-between** statements. This is like Pascal's semicolon separator, rather than C/C++'s semicolon terminator. For example:

```
statement,  
statement,  
statement, // ERROR!
```

Is an error. The last statement should not have a comma. Block statements can be grouped using the parentheses. Again, the last statement in a group should not have a trailing comma.

```
statement,  
statement,  
(  
    statement,  
    statement  
)  
,  
statement
```

Outside the square brackets, block statements should be grouped. For example:

```
for_each(c.begin(), c.end(),  
    (  
        do_this(arg1),  
        do_that(arg1)  
    )  
)
```

Wrapping a comma operator chain around a parentheses pair blocks the interpretation as an argument separator. The reason for the exception for the square bracket operator is that the operator always takes exactly one argument, so it "transforms" any attempt at multiple arguments with a comma operator chain (and spits out an error for zero arguments).

## **if\_ Statement**

```
#include <boost/spirit/home/phoenix/statement/if.hpp>
```

We have seen the `if_` statement. The syntax is:

```
if_(conditional_expression)
[
    sequenced_statements
]
```

## **ifelse statement**

```
#include <boost/spirit/home/phoenix/statement/if.hpp>
```

The syntax is

```
if_(conditional_expression)
[
    sequenced_statements
]
.else_
[
    sequenced_statements
]
```

Take note that `else` has a leading dot and a trailing underscore: `.else_`

Example: This code prints out all the elements and appends "`> 5`", "`== 5`" or "`< 5`" depending on the element's actual value:

```
for_each(c.begin(), c.end(),
    if_(arg1 > 5)
    [
        cout << arg1 << " > 5\n"
    ]
.else_
[
    if_(arg1 == 5)
    [
        cout << arg1 << " == 5\n"
    ]
.else_
[
    cout << arg1 << " < 5\n"
]
)
);
```

Notice how the `if_else_` statement is nested.

## **switch\_ statement**

```
#include <boost/spirit/home/phoenix/statement/switch.hpp>
```

The syntax is:

```
switch_(integral_expression)
[
    case_<integral_value>(sequenced_statements),
    ...
    default_<integral_value>(sequenced_statements)
]
```

A comma separated list of cases, and an optional default can be provided. Note unlike a normal switch statement, cases do not fall through.

Example: This code prints out "one", "two" or "other value" depending on the element's actual value:

```
for_each(c.begin(), c.end(),
    switch_(arg1)
    [
        case_<1>(cout << val("one") << '\n'),
        case_<2>(cout << val("two") << '\n'),
        default_(cout << val("other value") << '\n')
    ]
);
```

## **while\_ Statement**

```
#include <boost/spirit/home/phoenix/statement/while.hpp>
```

The syntax is:

```
while_(conditional_expression)
[
    sequenced_statements
]
```

Example: This code decrements each element until it reaches zero and prints out the number at each step. A newline terminates the printout of each value.

```
for_each(c.begin(), c.end(),
(
    while_(arg1--)
    [
        cout << arg1 << ", "
    ],
    cout << val("\n")
)
);
```

## **dowhile Statement**

```
#include <boost/spirit/home/phoenix/statement/do_while.hpp>
```

The syntax is:

```
do_
[
    sequenced_statements
]
.what_(conditional_expression)
```

Again, take note that `while` has a leading dot and a trailing underscore: `.while_`

Example: This code is almost the same as the previous example above with a slight twist in logic.

```
for_each(c.begin(), c.end(),
(
    do_
    [
        cout << arg1 << ", "
    ]
    .while_(arg1--),
    cout << val("\n")
)
);
```

## for\_ Statement

```
#include <boost/spirit/home/phoenix/statement/for.hpp>
```

The syntax is:

```
for_(init_statement, conditional_expression, step_statement)
[
    sequenced_statements
]
```

It is again very similar to the C++ `for` statement. Take note that the `init_statement`, `conditional_expression` and `stepstatement` are separated by the comma instead of the semi-colon and each must be present (i.e. ``for(,,)`` is invalid). This is a case where the `nothing` actor can be useful.

Example: This code prints each element N times where N is the element's value. A newline terminates the printout of each value.

```
int iii;
for_each(c.begin(), c.end(),
(
    for_(ref(iii) = 0, ref(iii) < arg1, ++ref(iii))
    [
        cout << arg1 << ", "
    ],
    cout << val("\n")
)
);
```

As before, all these are lazily evaluated. The result of such statements are in fact composites that are passed on to STL's `for_each` function. In the viewpoint of `for_each`, what was passed is just a functor, no more, no less.



### Note

Unlike lazy functions and lazy operators, lazy statements always return void.

## try\_catch\_ Statement

```
#include <boost/spirit/home/phoenix/statement/tryCatch.hpp>
```

The syntax is:

```
try_
[
    sequenced_statements
]
.catch_<exception_type>()
[
    sequenced_statements
]
...
.catch_all
[
    sequenced_statement
]
```

Note the usual underscore after try and catch, and the extra parentheses required after the catch.

Example: The following code calls the (lazy) function `f` for each element, and prints messages about different exception types it catches.

```
try_
[
    f(arg1)
]
.catch_<runtime_error>()
[
    cout << val("caught runtime error or derived\n")
]
.catch_<exception>()
[
    cout << val("caught exception or derived\n")
]
.catch_all
[
    cout << val("caught some other type of exception\n")
]
```

## throw\_

```
#include <boost/spirit/home/phoenix/statement/throw.hpp>
```

As a natural companion to the try/catch support, the statement module provides lazy throwing and rethrowing of exceptions.

The syntax to throw an exception is:

```
throw_(exception_expression)
```

The syntax to rethrow an exception is:

```
throw_()
```

Example: This code extends the try/catch example, rethrowing exceptions derived from runtime\_error or exception, and translating other exception types to runtime\_errors.

```
try_
[
    f(arg1)
]
.catch_<runtime_error>()
[
    cout << val("caught runtime error or derived\n"),
    throw_()
]
.catch_<exception>()
[
    cout << val("caught exception or derived\n"),
    throw_()
]
.catch_all
[
    cout << val("caught some other type of exception\n"),
    throw_(runtime_error("translated exception"))
]
```

## Object

The Object module deals with object construction, destruction and conversion. The module provides "*lazy*" versions of C++'s object constructor, new, delete, static\_cast, dynamic\_cast, const\_cast and reinterpret\_cast.

## Construction

### Lazy constructors...

```
#include <boost/spirit/home/phoenix/object/construct.hpp>
```

Lazily construct an object from an arbitrary set of arguments:

```
construct<T>(ctor_arg1, ctor_arg2, ..., ctor_argN);
```

where the given parameters are the parameters to the constructor of the object of type T (This implies, that type T is expected to have a constructor with a corresponding set of parameter types.).

Example:

```
construct<std::string>(arg1, arg2)
```

Constructs a std::string from arg1 and arg2.



### Note

The maximum number of actual parameters is limited by the preprocessor constant PHOENIX\_COMPOSITE\_LIMIT. Note though, that this limit should not be greater than PHOENIX\_LIMIT. By default, PHOENIX\_COMPOSITE\_LIMIT is set to PHOENIX\_LIMIT (See [Actors](#)).

## New

### Lazy new...

```
#include <boost/spirit/home/phoenix/object/new.hpp>
```

Lazily construct an object, on the heap, from an arbitrary set of arguments:

```
new_<T>(ctor_arg1, ctor_arg2, ..., ctor_argN);
```

where the given parameters are the parameters to the constructor of the object of type T (This implies, that type T is expected to have a constructor with a corresponding set of parameter types.).

Example:

```
new_<std::string>(arg1, arg2) // note the spelling of new_ (with trailing underscore)
```

Creates a std::string from arg1 and arg2 on the heap.



### Note

Again, the maximum number of actual parameters is limited by the preprocessor constant PHOENIX\_COMPOSITE\_LIMIT. See the note above.

## Delete

### Lazy delete...

```
#include <boost/spirit/home/phoenix/object/delete.hpp>
```

Lazily delete an object, from the heap:

```
delete_(arg);
```

where arg is assumed to be a pointer to an object.

Example:

```
delete_<std::string>(arg1) // note the spelling of delete_ (with trailing underscore)
```

## Casts

### Lazy casts...

```
#include <boost/spirit/home/phoenix/object/static_cast.hpp>
#include <boost/spirit/home/phoenix/object/dynamic_cast.hpp>
#include <boost/spirit/home/phoenix/object/const_cast.hpp>
#include <boost/spirit/home/phoenix/object/reinterpret_cast.hpp>
```

The set of lazy C++ cast template functions provide a way of lazily casting an object of a certain type to another type. The syntax resembles the well known C++ casts. Take note however that the lazy versions have a trailing underscore.

```
static_cast_<T>(lambda_expression)
dynamic_cast_<T>(lambda_expression)
const_cast_<T>(lambda_expression)
reinterpret_cast_<T>(lambda_expression)
```

Example:

```
static_cast_<Base*>(&arg1)
```

Static-casts the address of `arg1` to a `Base*`.

## Scope

Up until now, the most basic ingredient is missing: creation of and access to local variables in the stack. When recursion comes into play, you will soon realize the need to have true local variables. It may seem that we do not need this at all since an unnamed lambda function cannot call itself anyway; at least not directly. With some sort of arrangement, situations will arise where a lambda function becomes recursive. A typical situation occurs when we store a lambda function in a [Boost.Function](#), essentially naming the unnamed lambda.

There will also be situations where a lambda function gets passed as an argument to another function. This is a more common situation. In this case, the lambda function assumes a new scope; new arguments and possibly new local variables.

This section deals with local variables and nested lambda scopes.

## Local Variables

```
#include <boost/spirit/home/phoenix/scope/local_variable.hpp>
```

We use an instance of:

```
actor<local_variable<Key> >
```

to represent a local variable. The local variable acts as an imaginary data-bin where a local, stack based data will be placed. `Key` is an arbitrary type that is used to identify the local variable. Example:

```
struct size_key;
actor<local_variable<size_key> > size;
```

## Predefined Local Variables

There are a few predefined instances of `actor<local_variable<Key> >` named `_a..._z` that you can already use. To make use of them, simply use the namespace `boost::phoenix::local_names`:

```
using namespace boost::phoenix::local_names;
```

## let

```
#include <boost/spirit/home/phoenix/scope/let.hpp>
```

You declare local variables using the syntax:

```
let(local-declarations)
[
    let-body
]
```

`let` allows 1..N local variable declarations (where N == `PHOENIX_LOCAL_LIMIT`). Each declaration follows the form:

```
local-id = lambda-expression
```



## Note

You can set `PHOENIX_LOCAL_LIMIT`, the predefined maximum local variable declarations in a `let` expression. By default, `PHOENIX_LOCAL_LIMIT` is set to `PHOENIX_LIMIT`.

Example:

```
let(_a = 123, _b = 456)
[
    _a + _b
]
```

## Reference Preservation

The type of the local variable assumes the type of the lambda- expression. Type deduction is reference preserving. For example:

```
let(_a = arg1, _b = 456)
```

`_a` assumes the type of `arg1`: a reference to an argument, while `_b` has type `int`.

Consider this:

```
int i = 1;

let(_a = arg1)
[
    cout << --_a << ' '
]
(i);

cout << i << endl;
```

the output of above is : 0 0

While with this:

```
int i = 1;

let(_a = val(arg1))
[
    cout << --_a << ' '
]
(i);

cout << i << endl;
```

the output is : 0 1

Reference preservation is necessary because we need to have L-value access to outer lambda-scopes (especially the arguments). `args` and `refs` are L-values. `vals` are R-values.

## Visibility

The scope and lifetimes of the local variables is limited within the let-body. `let` blocks can be nested. A local variable may hide an outer local variable. For example:

```
let(_x = 1, _y = "World")
[
    // _x here is an int: 1

    let(_x = "Hello") // hides the outer _x
    [
        cout << _x << _y // prints "Hello, World"
    ]
]
```

The RHS (right hand side lambda-expression) of each local-declaration cannot refer to any LHS local-id. At this point, the local-ids are not in scope yet; they will only be in scope in the let-body. The code below is in error:

```
let(
    _a = 1
    , _b = _a // Error: _a is not in scope yet
)
[
    // _a and _b's scope starts here
    /*. body .*/
]
```

However, if an outer let scope is available, this will be searched. Since the scope of the RHS of a local-declaration is the outer scope enclosing the let, the RHS of a local-declaration can refer to a local variable of an outer scope:

```
let(_a = 1)
[
  let(
    _a = 1
    , _b = _a // Ok. _a refers to the outer _a
  )
  [
    /*. body .*/
  ]
]
```

## lambda

```
#include <boost/spirit/home/phoenix/scope/lambda.hpp>
```

A lot of times, you'd want to write a lazy function that accepts one or more functions (higher order functions). STL algorithms come to mind, for example. Consider a lazy version of `stl::for_each`:

```
struct for_each_impl
{
  template <typename C, typename F>
  struct result
  {
    typedef void type;
  };

  template <typename C, typename F>
  void operator()(C& c, F f) const
  {
    std::for_each(c.begin(), c.end(), f);
  }
};

function<for_each_impl> const for_each = for_each_impl();
```

Notice that the function accepts another function, `f` as an argument. The scope of this function, `f`, is limited within the `operator()`. When `f` is called inside `std::for_each`, it exists in a new scope, along with new arguments and, possibly, local variables. This new scope is not at all related to the outer scopes beyond the `operator()`.

Simple syntax:

```
lambda
[
  lambda-body
]
```

Like `let`, local variables may be declared, allowing 1..N local variable declarations (where N == PHOENIX\_LOCAL\_LIMIT):

```
lambda(local-declarations)
[
  lambda-body
]
```

The same restrictions apply with regard to scope and visibility. The RHS (right hand side lambda-expression) of each local-declaration cannot refer to any LHS local-id. The local-ids are not in scope yet; they will be in scope only in the lambda-body:

```
lambda(
    _a = 1
    , _b = _a // Error: _a is not in scope yet
)
```

See [let Visibility](#) above for more information.

Example: Using our lazy `for_each` let's print all the elements in a container:

```
for_each(arg1, lambda[cout << arg1])
```

As far as the arguments are concerned (`arg1..argN`), the scope in which the lambda-body exists is totally new. The left `arg1` refers to the argument passed to `for_each` (a container). The right `arg1` refers to the argument passed by `std::for_each` when we finally get to call `operator()` in our `for_each_impl` above (a container element).

Yet, we may wish to get information from outer scopes. While we do not have access to arguments in outer scopes, what we still have is access to local variables from outer scopes. We may only be able to pass argument related information from outer lambda scopes through the local variables.



## Note

This is a crucial difference between `let` and `lambda`: `let` does not introduce new arguments; `lambda` does.

Another example: Using our lazy `for_each`, and a lazy `push_back`:

```
struct push_back_impl
{
    template <typename C, typename T>
    struct result
    {
        typedef void type;
    };

    template <typename C, typename T>
    void operator()(C& c, T& x) const
    {
        c.push_back(x);
    }
};

function<push_back_impl> const push_back = push_back_impl();
```

write a lambda expression that accepts:

1. a 2-dimensional container (e.g. `vector<vector<int> >`)
2. a container element (e.g. `int`)

and pushes-back the element to each of the `vector<int>`.

Solution:

```
for_each(arg1,
    lambda(_a = arg2)
    [
        push_back(arg1, _a)
    ]
)
```

Since we do not have access to the arguments of the outer scopes beyond the lambda-body, we introduce a local variable `_a` that captures the second outer argument: `arg2`. Hence: `_a = arg2`. This local variable is visible inside the lambda scope.

(See [lambda.cpp](#))

## Bind

*Binding* is the act of tying together a function to some arguments for deferred (lazy) evaluation. Named [Lazy functions](#) require a bit of typing. Unlike (unnamed) lambda expressions, we need to write a functor somewhere offline, detached from the call site. If you wish to transform a plain function, member function or member variable to a lambda expression, bind is your friend.



### Note

Take note that binders are monomorphic. Rather than binding functions, the preferred way is to write true generic and polymorphic [lazy-functions](#). However, since most of the time we are dealing with adaptation of existing code, binders get the job done faster.

There is a set of overloaded bind template functions. Each `bind(x)` function generates a suitable binder object, a [composite](#).

## Binding Functions

```
#include <boost/spirit/home/phoenix/bind/bind_function.hpp>
```

Example, given a function `foo`:

```
void foo(int n)
{
    std::cout << n << std::endl;
}
```

Here's how the function `foo` may be bound:

```
bind(&foo, arg1)
```

This is now a full-fledged [composite](#) that can finally be evaluated by another function call invocation. A second function call will invoke the actual `foo` function. Example:

```
int i = 4;
bind(&foo, arg1)(i);
```

will print out "4".

## Binding Member Functions

```
#include <boost/spirit/home/phoenix/bind/bind_member_function.hpp>
```

Binding member functions can be done similarly. A bound member function takes in a pointer or reference to an object as the first argument. For instance, given:

```
struct xyz
{
    void foo(int) const;
};
```

`xyz`'s `foo` member function can be bound as:

```
bind(&xyz::foo, obj, arg1) // obj is an xyz object
```

Take note that a lazy-member functions expects the first argument to be a pointer or reference to an object. Both the object (reference or pointer) and the arguments can be lazily bound. Examples:

```
xyz obj;
bind(&xyz::foo, arg1, arg2)      // arg1.foo(arg2)
bind(&xyz::foo, obj, arg1)       // obj.foo(arg1)
bind(&xyz::foo, obj, 100)        // obj.foo(100)
```

## Binding Member Variables

```
#include <boost/spirit/home/phoenix/bind/bind_member_variable.hpp>
```

Member variables can also be bound much like member functions. Member variables are not functions. Yet, like the `ref(x)` that acts like a nullary function returning a reference to the data, member variables, when bound, act like a unary function, taking in a pointer or reference to an object as its argument and returning a reference to the bound member variable. For instance, given:

```
struct xyz
{
    int v;
};
```

`xyz::v` can be bound as:

```
bind(&xyz::v, obj) // obj is an xyz object
```

As noted, just like the bound member function, a bound member variable also expects the first (and only) argument to be a pointer or reference to an object. The object (reference or pointer) can be lazily bound. Examples:

```
xyz obj;
bind(&xyz::v, arg1)           // arg1.v
bind(&xyz::v, obj)           // obj.v
bind(&xyz::v, arg1)(obj) = 4 // obj.v = 4
```

## Container

```
#include <boost/spirit/home/phoenix/container.hpp>
```

The container module predefines a set of lazy functions that work on STL containers. These functions provide a mechanism for the lazy evaluation of the public member functions of the STL containers. The lazy functions are thin wrappers that simply forward to their respective counterparts in the STL library.

Lazy functions are provided for all of the member functions of the following containers:

- deque
- list
- map
- multimap
- vector

Indeed, should your class have member functions with the same names and signatures as those listed below, then it will automatically be supported. To summarize, lazy functions are provided for member functions:

- assign
- at
- back
- begin
- capacity
- clear
- empty
- end
- erase
- front
- get\_allocator
- insert
- key\_comp
- max\_size
- pop\_back
- pop\_front

- push\_back
- push\_front
- rbegin
- rend
- reserve
- resize
- size
- splice
- value\_comp

The lazy functions' names are the same as the corresponding member function. The difference is that the lazy functions are free functions and therefore does not use the member "dot" syntax.

**Table 5. Sample usage**

"Normal" version	"Lazy" version
<code>my_vector.at(5)</code>	<code>at(arg1, 5)</code>
<code>my_list.size()</code>	<code>size(arg1)</code>
<code>my_vector1.swap(my_vector2)</code>	<code>swap(arg1, arg2)</code>

Notice that member functions with names that clash with stl algorithms are absent. This will be provided in Phoenix's algorithm module.

No support is provided here for lazy versions of `operator+=`, `operator[]` etc. Such operators are not specific to STL containers and lazy versions can therefore be found in [operators](#).

The following table describes the container functions and their semantics.



Arguments in brackets denote optional parameters.

**Table 6. Lazy STL Container Functions**

Function	Semantics
assign(c, a[, b, c])	c.assign(a[, b, c])
at(c, i)	c.at(i)
back(c)	c.back()
begin(c)	c.begin()
capacity(c)	c.capacity()
clear(c)	c.clear()
empty(c)	c.empty()
end(c)	c.end()
erase(c, a[, b])	c.erase(a[, b])
front(c)	c.front()
get_allocator(c)	c.get_allocator()
insert(c, a[, b, c])	c.insert(a[, b, c])
key_comp(c)	c.key_comp()
max_size(c)	c.max_size()
pop_back(c)	c.pop_back()
pop_front(c)	c.pop_front()
push_back(c, d)	c.push_back(d)
push_front(c, d)	c.push_front(d)
pop_front(c)	c.pop_front()
rbegin(c)	c.rbegin()
rend(c)	c.rend()
reserve(c, n)	c.reserve(n)
resize(c, a[, b])	c.resize(a[, b])
size(c)	c.size()
splice(c, a[, b, c, d])	c.splice(a[, b, c, d])
value_comp(c)	c.value_comp()

# Algorithm

```
#include <boost/spirit/home/phoenix/algorithm.hpp>
```

The algorithm module provides wrappers for the standard algorithms in the `<algorithm>` and `<numeric>` headers.

The algorithms are divided into the categories iteration, transformation and querying, modelling the [Boost.MPL](#) library. The different algorithm classes can be included using the headers:

```
#include <boost/spirit/home/phoenix/stl/algorithm/iteration.hpp>
#include <boost/spirit/home/phoenix/stl/algorithm/transformation.hpp>
#include <boost/spirit/home/phoenix/stl/algorithm/querying.hpp>
```

The functions of the algorithm module take ranges as arguments where appropriate. This is different to the standard library, but easy enough to pick up. Ranges are described in detail in the [Boost.Range](#) library.

For example, using the standard copy algorithm to copy between 2 arrays:

```
int array[] = {1, 2, 3};
int output[3];
std::copy(array, array + 3, output); // We have to provide iterators
// to both the start and end of array
```

The analogous code using the phoenix algorithm module is:

```
int array[] = {1, 2, 3};
int output[3];
copy(arg1, arg2)(array, output); // Notice only 2 arguments, the end of
// array is established automatically
```

The [Boost.Range](#) library provides support for standard containers, strings and arrays, and can be extended to support additional types.

The following tables describe the different categories of algorithms, and their semantics.



Arguments in brackets denote optional parameters.

**Table 7. Iteration Algorithms**

Function	stl Semantics
<code>for_each(r, c)</code>	<code>for_each(begin(r), end(r), f)</code>
<code>accumulate(r, o[, f])</code>	<code>accumulate(begin(r), end(r), o[, f])</code>

**Table 8. Querying Algorithms**

Function	stl Semantics
<code>find(r, a)</code>	<code>find(begin(r), end(r), a)</code>
<code>find_if(r, f)</code>	<code>find_if(begin(r), end(r), f)</code>
<code>find_end(r1, r2[, f])</code>	<code>find_end(begin(r1), end(r1), begin(r2), end(r2)[, f])</code>
<code>find_first_of(r1, r2[, f])</code>	<code>find_first_of(begin(r1), end(r1), begin(r2), end(r2)[, f])</code>
<code>adjacent_find(r[, f])</code>	<code>adjacent_find(begin(r), end(r)[, f])</code>
<code>count(r, a)</code>	<code>count(begin(r), end(r), a)</code>
<code>count_if(r, f)</code>	<code>count_if(begin(r), end(r), f)</code>
<code>distance(r)</code>	<code>distance(begin(r), end(r))</code>
<code>mismatch(r, i[, f])</code>	<code>mismatch(begin(r), end(r), i[, f])</code>
<code>equal(r, i[, f])</code>	<code>equal(begin(r), end(r), i[, f])</code>
<code>search(r1, r2[, f])</code>	<code>search(begin(r1), end(r1), begin(r2), end(r2)[, f])</code>
<code>lower_bound(r, a[, f])</code>	<code>lower_bound(begin(r), end(r), a[, f])</code>
<code>upper_bound(r, a[, f])</code>	<code>upper_bound(begin(r), end(r), a[, f])</code>
<code>equal_range(r, a[, f])</code>	<code>equal_range(begin(r), end(r), a[, f])</code>
<code>binary_search(r, a[, f])</code>	<code>binary_search(begin(r), end(r), a[, f])</code>
<code>includes(r1, r2[, f])</code>	<code>includes(begin(r1), end(r1), begin(r2), end(r2)[, f])</code>
<code>min_element(r[, f])</code>	<code>min_element(begin(r), end(r)[, f])</code>
<code>max_element(r[, f])</code>	<code>max_element(begin(r), end(r)[, f])</code>
<code>lexicographical_compare(r1, r2[, f])</code>	<code>lexicographical_compare(begin(r1), end(r1), begin(r2), end(r2)[, f])</code>

**Table 9. Transformation Algorithms**

Function	stl Semantics
copy(r, o)	copy(begin(r), end(r), o)
copy_backward(r, o)	copy_backward(begin(r), end(r), o)
transform(r, o, f)	transform(begin(r), end(r), o, f)
transform(r, i, o, f)	transform(begin(r), end(r), i, o, f)
replace(r, a, b)	replace(begin(r), end(r), a, b)
replace_if(r, f, a)	replace(begin(r), end(r), f, a)
replace_copy(r, o, a, b)	replace_copy(begin(r), end(r), o, a, b)
replace_copy_if(r, o, f, a)	replace_copy_if(begin(r), end(r), o, f, a)
fill(r, a)	fill(begin(r), end(r), a)
fill_n(r, n, a)	fill_n(begin(r), n, a)
generate(r, f)	generate(begin(r), end(r), f)
generate_n(r, n, f)	generate_n(begin(r), n, f)
remove(r, a)	remove(begin(r), end(r), a)
remove_if(r, f)	remove_if(begin(r), end(r), f)
remove_copy(r, o, a)	remove_copy(begin(r), end(r), o, a)
remove_copy_if(r, o, f)	remove_copy_if(begin(r), end(r), o, f)
unique(r[, f])	unique(begin(r), end(r)[, f])
unique_copy(r, o[, f])	unique_copy(begin(r), end(r), o[, f])
reverse(r)	reverse(begin(r), end(r))
reverse_copy(r, o)	reverse_copy(begin(r), end(r), o)
rotate(r, m)	rotate(begin(r), m, end(r))
rotate_copy(r, m, o)	rotate_copy(begin(r), m, end(r), o)
random_shuffle(r[, f])	random_shuffle(begin(r), end(r), f)
partition(r, f)	partition(begin(r), end(r), f)
stable_partition(r, f)	stable_partition(begin(r), end(r), f)
sort(r[, f])	sort(begin(r), end(r)[, f])
stable_sort(r[, f])	stable_sort(begin(r), end(r)[, f])

Function	stl Semantics
partial_sort(r, m[, f])	partial_sort(begin(r), m, end(r)[, f])
partial_sort_copy(r1, r2[, f])	partial_sort_copy(begin(r1), end(r1), begin(r2), end(r2)[, f])
nth_element(r, n[, f])	nth_element(begin(r), n, end(r)[, f])
merge(r1, r2, o[, f])	merge(begin(r1), end(r1), begin(r2), end(r2), o[, f])
inplace_merge(r, m[, f])	inplace_merge(begin(r), m, end(r)[, f])
set_union(r1, r2, o[, f])	set_union(begin(r1), end(r1), begin(r2), end(r2)[, f])
set_intersection(r1, r2, o[, f])	set_intersection(begin(r1), end(r1), begin(r2), end(r2)[, f])
set_difference(r1, r2, o[, f])	set_difference(begin(r1), end(r1), begin(r2), end(r2)[, f])
set_symmetric_difference(r1, r2, o[, f])	set_symmetric_difference(begin(r1), end(r1), begin(r2), end(r2)[, f])
push_heap(r[, f])	push_heap(begin(r), end(r)[, f])
pop_heap(r[, f])	pop_heap(begin(r), end(r)[, f])
make_heap(r[, f])	make_heap(begin(r), end(r)[, f])
sort_heap(r[, f])	sort_heap(begin(r), end(r)[, f])
next_permutation(r[, f])	next_permutation(begin(r), end(r)[, f])
prev_permutation(r[, f])	prev_permutation(begin(r), end(r)[, f])
inner_product(r, o, a[, f1, f2])	inner_product(begin(r), end(r), o[, f1, f2])
partial_sum(r, o[, f])	partial_sum(begin(r), end(r), o[, f])
adjacent_difference(r, o[, f])	adjacent_difference(begin(r), end(r), o[, f])

## Inside Phoenix

This chapter explains in more detail how the library operates. The information henceforth should not be necessary to those who are interested in just using the library. However, a microscopic view might prove to be beneficial to moderate to advanced programmers who wish to extend the library.

## Actors In Detail

### Actor Concept

The main concept is the **Actor**. Actors are function objects (that can accept 0 to N arguments (where N is a predefined maximum).



## Note

You can set `PHOENIX_LIMIT`, the predefined maximum arity an actor can take. By default, `PHOENIX_LIMIT` is set to 10.

## actor template class

The `actor` template class models the `Actor` concept:

```
template <typename Eval>
struct actor : Eval
{
    typedef Eval eval_type;

    actor();
    actor(Eval const& base);

    template <typename T0>
    explicit actor(T0 const& _0);

    template <typename T0, typename T1>
    actor(T0 const& _0, T1 const& _1);

    // more constructors

    typename apply_actor<eval_type, basic_environment<> >::type
    operator()() const;

    template <typename T0>
    typename apply_actor<eval_type, basic_environment<T0> >::type
    operator()(T0& _0) const;

    template <typename T0, typename T1>
    typename apply_actor<eval_type, basic_environment<T0, T1> >::type
    operator()(T0& _0, T1& _1) const;

    // function call operators
};
```

**Table 10. Actor Concept Requirements**

Expression	Result/Semantics
<code>T::eval_type</code>	The actor's <code>Eval</code> type
<code>T()</code>	Default Constructor
<code>T(base)</code>	Constructor from <code>Eval</code>
<code>T(arg0, arg1, ..., argN)</code>	Pass through constructors
<code>x(arg0, arg1, ..., argN)</code>	Function call operators

## Eval Concept

The `actor` template class has a single template parameter, `Eval`, from which it derives from. While the `Actor` concept represents a function, the `Eval` concept represents the function body. The requirements for `Eval` are intentionally kept simple, to make it easy to write models of the concept. We shall see an example in the [next section](#).

**Table 11. Eval Concept Requirements**

Expression	Result/Semantics
<code>return x.eval(env)</code>	Evaluates the function (see Environment below)
<code>T::result&lt;Env&gt;::type</code>	The return type of eval (see Environment below)

## Constructors

In addition to a default constructor and an constructor from a Eval object, there are templated (pass through) constructors for 1 to N arguments ( $N == \text{PHOENIX\_LIMIT}$ ). These constructors simply forward the arguments to the base.



### Note

#### Parametric Base Class Pattern

Notice that actor derives from its template argument Eval. This is the inverse of the curiously recurring template pattern (CRTP). With the CRTP, a class, T, has a Derived template parameter that is assumed to be its subclass. The "parametric base class pattern" (PBCP), on the other hand, inverses the inheritance and makes a class, T, the derived class. Both CRTP and PBCP techniques have its pros and cons, which is outside the scope of this document. CRTP should really be renamed "parametric subclass pattern (PSCP)", but again, that's another story.

## Function Call Operators

There are N function call operators for 0 to N arguments ( $N == \text{PHOENIX\_LIMIT}$ ). The actor class accepts the arguments and forwards the arguments to the actor's base Eval for evaluation.



### Note

#### Forwarding Function Problem

The function call operators cannot accept non-const temporaries and literal constants. There is a known issue with current C++ called the "[Forwarding Function Problem](#)". The problem is that given an arbitrary function F, using current C++ language rules, one cannot create a forwarding function FF that transparently assumes the arguments of F. Disallowing non-const rvalues arguments partially solves the problem but prohibits code such as `f(1, 2, 3);`.

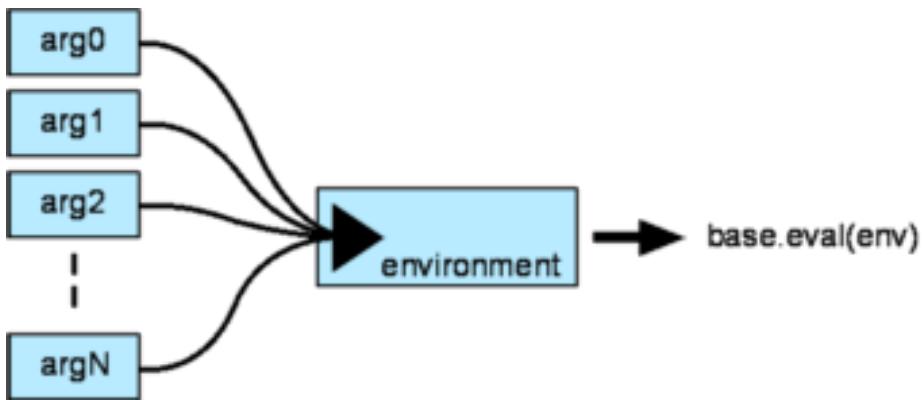
## Environment

On an actor function call, before calling the actor's `Eval::eval` for evaluation, the actor creates an *environment*. Basically, the environment packages the arguments in a tuple. The Environment is a concept, of which, the `basic_environment` template class is a model of.

**Table 12. Environment Concept Requirements**

Expression	Result/Semantics
<code>x.args()</code>	The arguments in a tie (a tuple of references)
<code>T::args_type</code>	The arguments' types in an MPL sequence
<code>T::tie_type</code>	The tie (tuple of references) type

Schematically:



Other parts of the library (e.g. the scope module) extends the Environment concept to hold other information such as local variables, etc.

## apply\_actor

`apply_actor` is a standard MPL style metafunction that simply calls the Action's `result` nested metafunction:

```

template <typename Action, typename Env>
struct apply_actor
{
    typedef typename Action::template result<Env>::type type;
};
  
```

After evaluating the arguments and doing some computation, the `eval` member function returns something back to the client. To do this, the forwarding function (the actor's `operator()`) needs to know the return type of the `eval` member function that it is calling. For this purpose, models of `Eval` are required to provide a nested template class:

```

template <typename Env>
struct result;
  
```

This nested class provides the result type information returned by the `Eval`'s `eval` member function. The nested template class `result` should have a `typedef type` that reflects the return type of its member function `eval`.

For reference, here's a typical `actor::operator()` that accepts two arguments:

```

template <typename T0, typename T1>
typename apply_actor<eval_type, basic_environment<T0, T1> >::type
operator()(T0& _0, T1& _1) const
{
    return eval_type::eval(basic_environment<T0, T1>(_0, _1));
}
  
```

## actor\_result

For reasons of symmetry to the family of `actor::operator()` there is a special metafunction usable for actor result type calculation named `actor_result`. This metafunction allows us to directly specify the types of the parameters to be passed to the `actor::operator()` function. Here's a typical `actor_result` that accepts two arguments:

```
template <typename Action, typename T0, typename T1>
struct actor_result
{
    typedef basic_environment<T0, T1> env_type;
    typedef typename Action::template result<env_type>::type type;
};
```

## Actor Example

Let us see a very simple prototypical example of an actor. This is not a toy example. This is actually part of the library. Remember the [reference](#)?

First, we have a model of the `Eval` concept: the `reference`:

```
template <typename T>
struct reference
{
    template <typename Env>
    struct result
    {
        typedef T& type;
    };

    reference(T& arg)
        : ref(arg) {}

    template <typename Env>
    T& eval(Env const&) const
    {
        return ref;
    }

    T& ref;
};
```

Models of `Eval` are never created directly and its instances never exist alone. We have to wrap it inside the `actor` template class to be useful. The `ref` template function does this for us:

```
template <typename T>
actor<reference<T> > const
ref(T& v)
{
    return reference<T>(v);
}
```

The `reference` template class conforms to the `Eval` concept. It has a nested `result` metaprogram that reflects the return type of its `eval` member function, which performs the actual function. `reference<T>` stores a reference to a `T`. Its `eval` member function simply returns the reference. It does not make use of the environment `Env`.

*Pretty simple...*

## Composites In Detail

We stated before that composites are actors that are composed of zero or more actors (see [Composite](#)). This is not quite accurate. The definition was sufficient at that point where we opted to keep things simple and not bury the reader with details which she might not need anyway.

Actually, a composite is a model of the [Eval](#) concept (more on this later). At the same time, it is also composed of 0..N (where N is a predefined maximum) [Eval](#) instances and an eval policy. The individual [Eval](#) instances are stored in a tuple.



## Note

In a sense, the original definition of "composite", more or less, will do just fine because [Eval](#) instances never exist alone and are always wrapped in an `actor` template class which inherits from it anyway. The resulting actor IS-AN [Eval](#).



## Note

You can set `PHOENIX_COMPOSITE_LIMIT`, the predefined maximum `Evals` (actors) a composite can take. By default, `PHOENIX_COMPOSITE_LIMIT` is set to `PHOENIX_LIMIT` (See [Actors](#)).

# composite template class

```
template <typename EvalPolicy, typename EvalTuple>
struct composite : EvalTuple
{
    typedef EvalTuple base_type;
    typedef EvalPolicy eval_policy_type;

    template <typename Env>
    struct result
    {
        typedef implementation-defined type;
    };

    composite();
    composite(base_type const& actors);

    template <typename U0>
    composite(U0 const& _0);

    template <typename U0, typename U1>
    composite(U0 const& _0, U1 const& _1);

    // more constructors

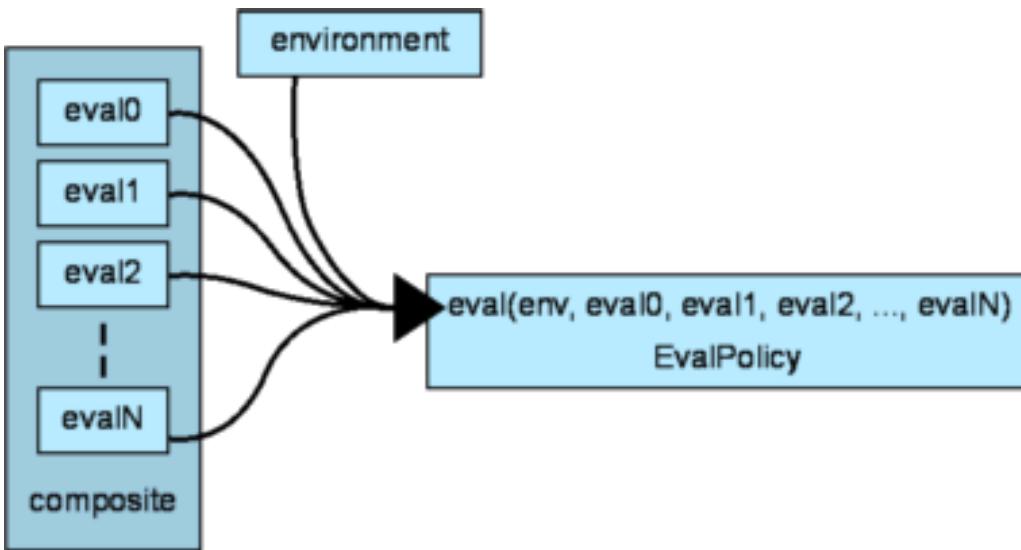
    template <typename Env>
    typename result<Env>::type
    eval(Env const& env) const;
};
```

## [EvalTuple](#)

`EvalTuple`, holds all the [Eval](#) instances. The `composite` template class inherits from it. In addition to a default constructor and a constructor from an `EvalTuple` object, there are templated (pass through) constructors for 1 to N arguments (again, where N == `PHOENIX_COMPOSITE_LIMIT`). These constructors simply forward the arguments to the `EvalTuple` base class.

## [EvalPolicy](#)

The composite's `eval` member function calls its `EvalPolicy`'s `eval` member function (a static member function) passing in the `environment` and each of its actors, in parallel. The following diagram illustrates what's happening:

**Table 13. EvalPolicy Requirements**

Expression	Result/Semantics
<code>x.eval&lt;RT&gt;(env, eval0, eval1, ..., evalN)</code>	Evaluate the composite
<code>T::result&lt;Env, Eval0, Eval1, Eval2, ..., EvalN&gt;::type</code>	The return type of eval

The `EvalPolicy` is expected to have a nested template class `result` which has a typedef `type` that reflects the return type of its member function `eval`. Here's a typical example of the composite's `eval` member function for a 2-actor composite:

```

template <typename Env>
typename result<Env>::type
eval(Env const& env) const
{
    typedef typename result<Env>::type return_type;
    return EvalPolicy::template
        eval<return_type>(
            env
            , get<0>(*this) // gets the 0th element from EvalTuple
            , get<1>(*this)); // gets the 1st element from EvalTuple
}

```

## Composing

Composites are never instantiated directly. Front end expression templates are used to generate the composites. Using expression templates, we implement a DSEL (Domain Specific Embedded Language) that mimicks native C++. You've seen this DSEL in action in the preceding sections. It is most evident in the [Statement](#) section.

There are some facilities in the library to make composition of composites easier. We have a set of overloaded `compose` functions and an `as_composite` metafunction. Together, these helpers make composing a breeze. We'll provide an example of a composite later to see why.

## compose

```
compose<EvalPolicy>(arg0, arg1, arg2, ..., argN);
```

Given an `EvalPolicy` and some arguments `arg0...argN`, returns a proper composite. The arguments may or may not be phoenix actors (primitives of composites). If not, the arguments are converted to actors appropriately. For example:

```
compose<X>( 3 )
```

converts the argument 3 to an `actor<value<int> >(3)`.

## as\_composite

```
as_composite<EvalPolicy, Arg0, Arg1, Arg2, ..., ArgN>::type
```

This is the metafunction counterpart of `compose`. Given an `EvalPolicy` and some argument types `Arg0...ArgN`, returns a proper composite type. For example:

```
as_composite<X, int>::type
```

is the composite type of the `compose<X>( 3 )` expression above.

## Composite Example

Now, let's examine an example. Again, this is not a toy example. This is actually part of the library. Remember the `while_` lazy statement? Putting together everything we've learned so far, we will present it here in its entirety (verbatim):

```

struct while_eval
{
    template <typename Env, typename Cond, typename Do>
    struct result
    {
        typedef void type;
    };

    template <typename RT, typename Env, typename Cond, typename Do>
    static void
    eval(Env const& env, Cond& cond, Do& do_)
    {
        while (cond.eval(env))
            do_.eval(env);
    }
};

template <typename Cond>
struct while_gen
{
    while_gen(Cond const& cond)
        : cond(cond) {}

    template <typename Do>
    actor<typename as_composite<while_eval, Cond, Do>::type>
    operator[](Do const& do_) const
    {
        return compose<while_eval>(cond, do_);
    }

    Cond cond;
};

template <typename Cond>
while_gen<Cond>
while_(Cond const& cond)
{
    return while_gen<Cond>(cond);
}

```

`while_eval` is an example of an [EvalPolicy](#). `while_gen` and `while_` are the expression template front ends. Let's break this apart to understand what's happening. Let's start at the bottom. It's easier that way.

When you write:

```
while_(cond)
```

we generate an instance of `while_gen<Cond>`, where `Cond` is the type of `cond`. `cond` can be an arbitrarily complex actor expression. The `while_gen` template class has an `operator[]` accepting another expression. If we write:

```
while_(cond)
[
    do_
]
```

it will generate a proper composite with the type:

```
as_composite<while_eval, Cond, Do>::type
```

where `Cond` is the type of `cond` and `Do` is the type of `do_`. Notice how we are using phoenix's [composition](#) (`compose` and `as_composite`) mechanisms here

```
template <typename Do>
actor<typename as_composite<while_eval, Cond, Do>::type>
operator[](Do const& do_) const
{
    return compose<while_eval>(cond, do_);
}
```

Finally, the `while_eval` does its thing:

```
while (cond.eval(env))
    do_.eval(env);
```

`cond` and `do_`, at this point, are instances of [Eval](#). `cond` and `do_` are the [Eval](#) elements held by the composite's [EvalTuple](#). `env` is the [Environment](#).

## Extending

We've shown how it is very easy to extend phoenix by writing new primitives and composites. The modular design of Phoenix makes it extremely extensible. We have seen that layer upon layer, the whole library is built on a solid foundation. There are only a few simple well designed concepts that are laid out like bricks. Overall, the library is designed to be extended. Everything above the core layer can in fact be considered just as extensions to the library. This modular design was inherited from the [Spirit](#) inline parser library.

Extension is non-intrusive. And, whenever a component or module is extended, the new extension automatically becomes a first class citizen and is automatically recognized by all modules and components in the library.

## Wrap Up

Sooner or later more FP techniques become standard practice as people find the true value of this programming discipline outside the academe and into the mainstream. In as much as structured programming of the 70s and object oriented programming in the 80s and generic programming in the 90s shaped our thoughts towards a more robust sense of software engineering, FP will certainly be a paradigm that will catapult us towards more powerful software design and engineering onward into the new millenium.

Let me quote Doug Gregor of Boost.org. About functional style programming libraries:

*They're gaining acceptance, but are somewhat stunted by the ubiquitousness of broken compilers. The C++ community is moving deeper into the so-called "STL-style" programming paradigm, which brings many aspects of functional programming into the fold. Look at, for instance, the Spirit parser to see how such function objects can be used to build Yacc-like grammars with semantic actions that can build abstract syntax trees on the fly. This type of functional composition is gaining momentum.*

Indeed. Phoenix is another attempt to introduce more FP techniques into the mainstream. Not only is it a tool that will make life easier for the programmer. In its own right, the actual design of the library itself is a model of true C++ FP in action. The library is designed and structured in a strict but clear and well mannered FP sense. By all means, use the library as a tool. But for those who want to learn more about FP in C++, don't stop there, I invite you to take a closer look at the design of the library itself.

So there you have it. Have fun! See you in the FP world.

## Acknowledgement

1. Hartmut Kaiser implemented the original lazy casts and constructors based on his original work on Spirit SE "semantic expressions" (the precursor to Phoenix).
2. Angus Leeming implemented the container functions on Phoenix-1 which I then ported to Phoenix-2.
3. Daniel Wallin helped with the scope module, local variables, let and lambda and the algorithms. I frequently discuss design issues with Daniel on Yahoo Messenger.
4. Jaakko Jarvi. DA Lambda MAN!
5. Dave Abrahams, for his constant presence, wherever, whenever.
6. Aleksey Gurtovoy, DA MPL MAN!
7. Doug Gregor, always a source of inspiration.
8. Dan Marsden, did almost all the work in bringing Phoenix-2 out the door.
9. Eric Niebler did a 2.0 pre-release review and wrote some range related code that Phoenix stole and used in the algorithms.
10. Thorsten Ottosen; Eric's range\_ex code began life as "container\_algo" in the old boost sandbox, by Thorsten in 2002-2003.
11. Jeremy Siek, even prior to Thorsten, in 2001, started the "container\_algo".
12. Vladimir Prus wrote the mutating algorithms code from the Boost Wiki.
13. Daryle Walker did a 2.0 pre-release review.

## References

1. Why Functional Programming Matters, John Hughes, 1989. Available online at <http://www.math.chalmers.se/~rjmh/Papers/why-fp.html>.
2. Boost.Lambda library, Jaakko Jarvi, 1999-2004 Jaakko Jarvi, Gary Powell. Available online at <http://www.boost.org/libs/lambda/>.
3. Functional Programming in C++ using the FC++ Library: a short article introducing FC++, Brian McNamara and Yannis Smaragdakis, August 2003. Available online at <http://www.cc.gatech.edu/~yannis/fc++/>.
4. Side-effects and partial function application in C++, Jaakko Jarvi and Gary Powell, 2001. Available online at <http://osl.iu.edu/~ja-jarvi/publications/papers/mpool01.pdf>.
5. Spirit Version 1.8.1, Joel de Guzman, Nov 2004. Available online at <http://www.boost.org/libs/spirit/>.
6. The Boost MPL Library, Aleksey Gurtovoy and David Abrahams, 2002-2004. Available online at <http://www.boost.org/libs/mpl/>.
7. Generic Programming Redesign of Patterns, Proceedings of the 5th European Conference on Pattern Languages of Programs, (EuroPLoP'2000) Irsee, Germany, July 2000. Available online at <http://www.coldewey.com/europlop2000/papers/geraud%2Bduret.zip>.
8. A Gentle Introduction to Haskell, Paul Hudak, John Peterson and Joseph Fasel, 1999. Available online at <http://www.haskell.org/tutorial/>.
9. Large scale software design, John Lackos, ISBN 0201633620, Addison-Wesley, July 1996.
10. Design Patterns, Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995.

11. The Forwarding Problem: Arguments Peter Dimov, Howard E. Hinnant, Dave Abrahams, September 09, 2002. Available online: [Forwarding Function Problem](#).

