Phoenix 3.0

Joel de Guzman

Dan Marsden

Thomas Heller

Table of Contents

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 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.

 Unless otherwise noted `using namespace boost::phoenix;` is assumed

...To Joel's dear daughter, Phoenix

What's New

Phoenix 3.0

This is the first official release of Phoenix as first class Boost citizen. As a consequence of the review of Phoenix V2 the internals got completely rewritten. Therefore the internal extension mechanism is different.

- composite<...>, as_composite<...> and compose are gone and have been replaced. For an in depth discussion see the section Inside Phoenix
- phoenix.modules.function phoenix::function now supports function objects that implement the Boost.Result Of protocol. **This is a breaking change**
- Boilerplate macros to easily adapt already existing functions and function objects
- Bind is no completely compatible with Boost.Bind

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 section 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 Documentation. 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 examples 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.

We start with some core functions that 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 **expressions**. The expressions can, in turn, be composed to form even more complex expressions.

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` is a unary function and `val(3)` invokes it, you say? Yes. 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:

```
std::cout << val(3)() << std::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";
std::cout << arg1(i) << std::endl;        // prints 3
std::cout << arg2(i, s) << std::endl;     // prints "Hello World"
```

(See arguments.cpp)

 Learn more about arguments here.

Lazy Operators

You can use the usual set of operators to form expressions. 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 expression.
- In a unary expression (e.g. `arg1++`), the single operand must be a phoenix primitive or expression.

If these basic rules are not followed, the result is either an 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 expression)
ref(x[ref(i)])  // illegal (x is not a phoenix primitive or expression)
```

Why are the last two expression illegal? Although `operator[]` looks as much like a binary operator as `operator=` above it; the difference is that the former must be a member (i.e. `x` must have an `operator[]` that takes a phoenix primitive or expression as its argument). This will most likely not be the case.

 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:

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

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

```
std::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)
[
    std::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:

```
std::for_each(v.begin(), v.end(),
    if_(arg1 > 5)
    [
        std::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
{
    typedef bool result_type;

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

function<is_odd_impl> is_odd;
```

Things to note:

- Result type deduction is implemented with the help of the result_of protocol. For more information see Boost.Result Of
- `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:

```
std::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 STL).

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, ..., a/n/)`, 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. 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. And even more, with Phoenix, these resulting functions won't be black boxes anymore.

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:

```
std::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;

std::cout << (arg1 % 2 == 1)(x) << std::endl; // prints 1 or true
std::cout << (arg1 % 2 == 1)(y) << std::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`.

In Phoenix we emulated perfect forwarding through preprocessor macros generating code to allow const and non-const references.

We generate these second-stage overloads for Phoenix expression up to `BOOST_PHOENIX_PERFECT_FORWARD_LIMIT`

Note

You can set `BOOST_PHOENIX_PERFECT_FORWARD_LIMIT`, the predefined maximum perfect forward arguments an actor can take. By default, `BOOST_PHOENIX_PERFECT_FORWARDLIMIT` is set to 3.

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:

1. Actor
2. Value, Reference, Arguments
3. Function, Operator, Object, Statement, Scope
4. STL, Fusion, Bind

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 desirable 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.

Terminals provide the basic building blocks of functionality within Phoenix. Expressions are used to combine these terminals together to provide more powerful functionality.

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

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.
STL Container	Set of predefined "lazy" functions that work on STL containers and sequences (e.g. `push_back`).
STL 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/phoenix/core.hpp>`.

Table 3. Includes

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

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

Actor

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

Note

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

Phoenix supplies an `actor` class template whose specializations model the `Actor` concept. `actor` has one template parameter, `Expr`, that supplies the underlying expression to evaluate.

```
template <typename Expr>
struct actor
{
    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 `BOOST_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 passed through to the evaluation mechanism. For more information see Inside Actors.

Modules

Core

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, these are all defined in the Core module.

This section shall deal with these preset primitives.

Values

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

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

```
expression::value<T>::type
```

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

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

Passing a second argument, `6`, an `expression::value<T>::type` is implicitly created behind the scenes. This is also equivalent to `add(arg1, val(6))`.

```
val(v)
```

generates an `expression::value<T>::type` 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/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 to be immutable values (see Values). To achieve our intent, we use:

```
expression::reference<T>::type
```

This is similar to `expression::value<T>::type` before but instead holds a reference to a variable.

We normally don't instantiate `expression::reference<T>::type` objects directly. Instead we use:

```
ref(v)
```

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 reference's identity. Example:

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

prints out "3 Hello World"

Constant References

Another free function

```
cref(cv)
```

may also be used. `cref(cv)` creates an `expression::reference<T const>::type` object. This is similar to `expression::value<T>::type` 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.

Arguments

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

We use an instance of:

```
expression::argument<N>::type
```

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 `expression::argument<N>::type` named `arg1`..`argN`, and its BLL counterpart `_1`..`_N`. (where N is a predefined maximum).

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

```
namespace placeholders
{
    expression::argument<1>::type const arg1 = {};
    expression::argument<2>::type const arg2 = {};
    expression::argument<3>::type const arg3 = {};
}
```

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

```
namespace placeholders
{
    expression::argument<1>::type const _1 = {};
    expression::argument<2>::type const _2 = {};
    expression::argument<3>::type const _3 = {};
}
```

Note

You can set `BOOST_PHOENIX_ARG_LIMIT`, the predefined maximum placeholder index. By default, `BOOST_PHOENIX_ARG_LIMIT` is set to `BOOST_PHOENIX_LIMIT` (See Actor).

User Defined Arguments

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

```
expression::argument<1>::type x; // note one 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 desirable. 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.

Nothing

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

Finally, the `expression::null<mpl::void_>::type` does nothing; (a "bum", if you will :-)). There's a sole `expression::null<mpl::void_>::type` instance named "nothing". This actor is actually useful in situations where we don't want to do anything. (See for_ Statement for example).

Function

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.

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

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 Polymorphic Function Object concept. For a function that takes `N` arguments, a model of Polymorphic Function Object must provide:

- An `operator()` that takes `N` arguments, and implements the function logic. This is also true for ordinary function pointers.
- A nested metafunction `result<Signature>` or nested typedef `result_type`, following the Boost.Result Of Protocol

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

```
struct factorial_impl
{
    template <typename Sig>
    struct result;

    template <typename This, typename Arg>
    struct result<This(Arg const &)>
    {
        typedef Arg type;
    };

    template <typename Arg>
    Arg operator()(Arg const & n) const
    {
        return (n <= 0) ? 1 : n * (*this)(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:

```
std::cout << factorial(arg1)(4);
```

will print out "24".

Take note that in certain cases (e.g. for function objects with state), an instance of the model of Polymorphic Function Object 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).

Important

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.

Adapting Functions

If you want to adapt already existing functions or function objects it will become a repetetive task. Therefor the following boilerplate macros are provided to help you adapt already exsiting functions, thus reducing the need to phoenix.modules.bind functions.

BOOST_PHOENIX_ADAPT_FUNCTION_NULLARY

Description

`BOOST_PHOENIX_ADAPT_FUNCTION_NULLARY` is a macro that can be used to generate all the necessary boilerplate to make an arbitrary nullary function a lazy function.

Note

These macros generate no global objects. The resulting lazy functions are real functions that create the lazy function expression object

Synopsis

```
BOOST_PHOENIX_ADAPT_FUNCTION_NULLARY(
    RETURN_TYPE
  , LAZY_FUNCTION
  , FUNCTION
)
```

Semantics

The above macro generates all necessary code to have a nullary lazy function `LAZY_FUNCTION` which calls the nullary `FUNCTION` that has the return type `RETURN_TYPE`

Header

```
#include <boost/phoenix/function/adapt_function.hpp>
```

Example

```
namespace demo
{
    int foo()
    {
        return 42;
    }
}

BOOST_PHOENIX_ADAPT_FUNCTION_NULLARY(int, foo, demo::foo)

int main()
{
    using boost::phoenix::placeholders::_1;

    assert((_1 + foo())(1) == 43);
}
```

BOOST_PHOENIX_ADAPT_FUNCTION

Description

`BOOST_PHOENIX_ADAPT_FUNCTION` is a macro that can be used to generate all the necessary boilerplate to make an arbitrary function a lazy function.

Synopsis

```
BOOST_PHOENIX_ADAPT_FUNCTION(
    RETURN_TYPE
  , LAZY_FUNCTION
  , FUNCTION
  , FUNCTION_ARITY
)
```

Semantics

The above macro generates all necessary code to have a lazy function `LAZY_FUNCTION` which calls `FUNCTION` that has the return type `RETURN_TYPE` with `FUNCTION_ARITY` number of arguments.

Header

```
#include <boost/phoenix/function/adapt_function.hpp>
```

Example

```
namespace demo
{
    int plus(int a, int b)
    {
        return a + b;
    }

    template <typename T>
    T
    plus(T a, T b, T c)
    {
        return a + b + c;
    }
}

BOOST_PHOENIX_ADAPT_FUNCTION(int, plus, demo::plus, 2)

BOOST_PHOENIX_ADAPT_FUNCTION(
    typename remove_reference<A0>::type
  , plus
  , demo::plus
  , 3
)

int main()
{
    using boost::phoenix::arg_names::arg1;
    using boost::phoenix::arg_names::arg2;

    int a = 123;
    int b = 256;

    assert(plus(arg1, arg2)(a, b) == a+b);
    assert(plus(arg1, arg2, 3)(a, b) == a+b+3);
}
```

BOOST_PHOENIX_ADAPT_CALLABLE_NULLARY

Description

`BOOST_PHOENIX_ADAPT_CALLABLE_NULLARY` is a macro that can be used to generate all the necessary boilerplate to make an arbitrary nullary function object a lazy function.

Synopsis

```
BOOST_PHOENIX_ADAPT_CALLABLE_NULLARY(
    LAZY_FUNCTION
  , CALLABLE
)
```

Semantics

The above macro generates all necessary code to create `LAZY_FUNCTION` which creates a lazy function object that represents a nullary call to `CALLABLE`. The return type is specified by `CALLABLE` conforming to the Boost.Result Of protocol.

Header

```
#include <boost/phoenix/function/adapt_callable.hpp>
```

Example

```
namespace demo
{
    struct foo
    {
        typedef int result_type;

        result_type operator()() const
        {
            return 42;
        }
    }
}

BOOST_PHOENIX_ADAPT_CALLABLE_NULLARY(foo, demo::foo)

int main()
{
    using boost::phoenix::placeholders::_1;

    assert((_1 + foo())(1) == 43);
}
```

BOOST_PHOENIX_ADAPT_CALLABLE

Description

`BOOST_PHOENIX_ADAPT_CALLABLE` is a macro that can be used to generate all the necessary boilerplate to make an arbitrary function object a lazy function.

Synopsis

```
BOOST_PHOENIX_ADAPT_CALLABLE(
    LAZY_FUNCTION
  , FUNCTION_NAME
  , FUNCTION_ARITY
)
```

Semantics

The above macro generates all necessary code to create `LAZY_FUNCTION` which creates a lazy function object that represents a call to `CALLABLE` with `FUNCTION_ARITY` arguments. The return type is specified by `CALLABLE` conforming to the Boost.Result Of protocol.

Header

```
#include <boost/phoenix/function/adapt_callable.hpp>
```

Example

```
namespace demo
{
    struct plus
    {
        template <typename Sig>
        struct result;

        template <typename This, typename A0, typename A1>
        struct result<This(A0, A1)>
            : remove_reference<A0>
        {};

        template <typename This, typename A0, typename A1, typename A2>
        struct result<This(A0, A1, A2)>
            : remove_reference<A0>
        {};

        template <typename A0, typename A1>
        A0 operator()(A0 const & a0, A1 const & a1) const
        {
            return a0 + a1;
        }

        template <typename A0, typename A1, typename A2>
        A0 operator()(A0 const & a0, A1 const & a1, A2 const & a2) const
        {
            return a0 + a1 + a2;
        }
    };
}

BOOST_PHOENIX_ADAPT_CALLABLE(plus, demo::plus, 2)

BOOST_PHOENIX_ADAPT_CALLABLE(plus, demo::plus, 3)

int main()
{
    using boost::phoenix::arg_names::arg1;
    using boost::phoenix::arg_names::arg2;
```

```
    int a = 123;
    int b = 256;

    assert(plus(arg1, arg2)(a, b) == a+b);
    assert(plus(arg1, arg2, 3)(a, b) == a+b+3);
}
```

Operator

```
#include <boost/phoenix/operator.hpp>
```

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 - Lazy Operators). Let's go back and examine them a little bit further:

```
std::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 expression 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 Actor) 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:

```
std::cout << ((arg1 + arg2) * arg3)(4, 5, 6);
```

will print out "54".

Operator expressions are lazily evaluated following four simple rules:

1. A binary operator, except `->*` will be lazily evaluated when *at least* one of its operands is an actor object (see Actor).
2. Unary operators are lazily evaluated if their argument is an actor object.
3. Operator `->*` 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, 2 and 3.

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

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

1. Following rule 1, `arg1 + 3` is lazily evaluated since `arg1` is an actor (see Arguments).

2. The result of this `arg1 + 3` expression is an actor object, following rule 4.

3. Continuing, `arg1 + 3 + 6` is again lazily evaluated. Rule 2.

4. By rule 4 again, the result of `arg1 + 3 + 6` is an actor object.

5. As `arg1 + 3 + 6` is an actor, `-(arg1 + 3 + 6)` is lazily evaluated. Rule 2.

Lazy-operator application is highly contagious. In most cases, a single `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 Core). 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)
```

Include Files

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

Statement

Lazy statements...

The expressions 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):

```
std::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++!!!

Note

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

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

Block Statement

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:

```
std::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/phoenix/statement/if.hpp>
```

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

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

if_else_ Statement

```
#include <boost/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:

```
std::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/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:

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

while_ Statement

```
#include <boost/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.

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

do_while_ Statement

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

The syntax is:

```
do_
[
    sequenced_statements
]
.while_(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.

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

for_ Statement

```
#include <boost/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 step_statement 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;
std::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 expressions 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.

try_ catch_ Statement

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

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

The syntax to throw an exception is:

```
throw_(exception_expression)
```

The syntax to re-throw an exception is:

```
throw_()
```

Example: This code extends the try/catch example, re-throwing 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/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 BOOST_PHOENIX_COMPOSITE_LIMIT. Note though, that this limit should not be greater than BOOST_PHOENIX_LIMIT. By default, `BOOST_PHOENIX_COMPOSITE_LIMIT` is set to `BOOST_PHOENIX_LIMIT` (See Actor).

New

Lazy new...

```
#include <boost/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 contractor 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

The maximum number of actual parameters is limited by the preprocessor constant BOOST_PHOENIX_COMPOSITE_LIMIT. Note though, that this limit should not be greater than BOOST_PHOENIX_LIMIT. By default, `BOOST_PHOENIX_COMPOSITE_LIMIT` is set to `BOOST_PHOENIX_LIMIT` (See Actor).

Delete

Lazy delete...

```
#include <boost/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/phoenix/object/static_cast.hpp>
#include <boost/phoenix/object/dynamic_cast.hpp>
#include <boost/phoenix/object/const_cast.hpp>
#include <boost/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/phoenix/scope/local_variable.hpp>
```

We use an instance of:

```
expression::local_variable<Key>::type
```

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;
expression::local_variable<size_key>::type size;
```

Predefined Local Variables

There are a few predefined instances of `expression::local_variable<Key>::type` 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/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 == `BOOST_PHOENIX_LOCAL_LIMIT`). Each declaration follows the form:

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

Note

You can set `BOOST_PHOENIX_LOCAL_LIMIT`, the predefined maximum local variable declarations in a let expression. By default, `BOOST_PHOENIX_LOCAL_LIMIT` is set to `BOOST_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/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 == `BOOST_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 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 binding functions, member functions or member variables is monomorphic. Rather than binding functions, the preferred way is to write true generic and polymorphic lazy functions.

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

Binding Function Objects

```
#include <boost/phoenix/bind/bind_function_object.hpp>
```

Binding function objects serves two purposes: * Partial function application * Quick adaption of already existing function objects

In order to deduce the return type of the function object, it has to implement the Boost.Result Of protocol. If the bound function object is polymorphic, the resulting binding object is polymorphic.

Binding Functions

```
#include <boost/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 expression that can finally be evaluated by another function call invocation. A second function call will invoke the actual `foo` function. Example:

```
bind(&foo, arg1)(4);
```

will print out "4".

Binding Member Functions

```
#include <boost/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/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
```

Compatibility with Boost.Bind

`phoenix::bind` passes all testcases of the Boost.Bind library. It is therefore completely compatible and interchangeable.

Given the compatibility with Boost.Bind, we also assume compatibility with std::tr1::bind and std::bind from the upcoming C++0x standard.

STL

```
#include <boost/phoenix/stl.hpp>
```

This section summarizes the lazy equivalents of C++ Standard Library functionality

Container

```
#include <boost/phoenix/stl/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 4. Sample usage

"Normal" version	"Lazy" version
`my_vector.at(5)`	`at(arg1, 5)`
`my_list.size()`	`size(arg1)`
`my_vector1.swap(my_vector2)`	`swap(arg1, arg2)`

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 5. 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/phoenix/stl/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, modeling the Boost.MPL library. The different algorithm classes can be included using the headers:

```
#include <boost/phoenix/stl/algorithm/iteration.hpp>
#include <boost/phoenix/stl/algorithm/transformation.hpp>
#include <boost/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 6. Iteration Algorithms

Function	stl Semantics
`for_each(r, f)`	`for_each(begin(r), end(r), f)`
`accumulate(r, o[, f])`	`accumulate(begin(r), end(r), o[, f])`

Table 7. Querying Algorithms

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

Table 8. 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 advanced programmers who wish to extend the library.

Actors in Detail

Actor

The main concept is the `Actor`. An `Actor` is a model of the Polymorphic Function Object concept (that can accept 0 to N arguments (where N is a predefined maximum).

An `Actor` contains a valid Phoenix Expression, a call to one of the function call operator overloads, starts the evaluation process.

Note

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

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

```
template <typename Expr>
struct actor
{
    template <typename Sig>
    struct result;

    typename result_of::actor<Expr>::type
    operator()() const;

    template <typename T0>
    typename result_of::actor<Expr, T0 &>::type
    operator()(T0& _0) const;

    template <typename T0>
    typename result_of::actor<Expr, T0 const &>::type
    operator()(T0 const & _0) const;

    //...
};
```

Table 9. Actor Concept Requirements

Expression	Semantics
`actor(arg0, arg1, ..., argN)`	Function call operators to start the evaluation
`boost::result_of<Actor<Expr>(Arg0, Arg1, ..., ArgN)>::type`	Result of the evaluation
`result_of::actor<Expr, Arg0, Arg1, ..., ArgN>::type`	Result of the evaluation

Function Call Operators

There are 2*N function call operators for 0 to N arguments (N == `BOOST_PHOENIX_LIMIT`). The actor class accepts the arguments and forwards the arguments to the default evaluation action.

Additionally, there exist function call operators accepting permutations of const and non-const references. These operators are created for all N <= `BOOST_PHOENIX_PERFECT_FORWARD_LIMIT` (which defaults to 3).

Note

Forwarding Function Problem

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`.

Context

On an actor function call, before calling the evaluation function, the actor created a **context**. This context consists of an `Environment` and an `Action` part. These contain all information necessary to evaluate the given expression.

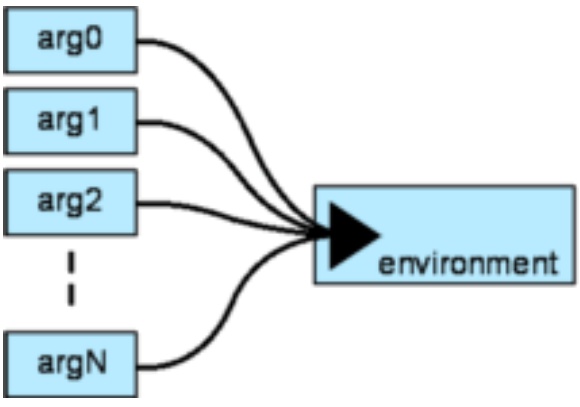
Table 10. Context Concept Requirements

Expression	Semantics
`result_of::context<Env, Actions>::type`	Type of a Context
`context(e, a)`	A Context containing environment `e` and actions `a`
`result_of::env<Context>::type`	Type of the contained Environment
`env(ctx)`	The environment
`result_of::actions<Context>::type`	Type of the contained Actions
`actions(ctx)`	The actions

Environment

The Environment is a model of Random Access Sequence.

The arguments passed to the actor's function call operator are collected inside the Environment:



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

Actions

Actions is the part of Phoenix which are responsible for giving the actual expressions a specific behaviour. During the traversal of the Phoenix Expression Tree these actions are called whenever a specified rule in the grammar matches.

```
struct actions
{
    template <typename Rule>
    struct when;
};
```

The nested `when` template is required to be Proto Primitive Transform. No worries, you don't have to learn Boost.Proto just yet! Phoenix provides some wrappers to let you define simple actions without the need to dive deep into proto.

Phoenix ships with a predefined `default_actions` class that evaluates the expressions with C++ semantics:

```
struct default_actions
{
    template <typename Rule, typename Dummy = void>
    struct when
        : proto::_default<meta_grammar>
    {};
};
```

For more information on how to use the default_actions class and how to attach custom actions to the evaluation process, see more on actions.

Evaluation

```
struct evaluator
{
    template <typename Expr, typename Context>
    unspecified operator()(Expr &, Context &);
};

evaluator const eval = {};
```

The evaluation of a Phoenix expression is started by a call to the function call operator of `evaluator`.

The evaluator is called by the `actor` function operator overloads after the context is built up. For reference, here is a typical `actor::operator()` that accepts two arguments:

```
template <typename T0, typename T1>
typename result_of::actor<Expr, T0 &, T1 &>::type
operator()(T0 &t0, T1 &t1) const
{
    fusion::vector2<T0 &, T1 &> env(t0, t1);

    return eval(*this, context(env, default_actions()));
}
```

result_of::actor

For reasons of symmetry to the family of `actor::operator()` there is a special metafunction usable for actor result type calculation named `result_of::actor`. 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:

```
namespace result_of
{
    template <typename Expr, typename T0, typename T1>
    struct actor
    {
        typedef fusion::vector2<T0, T1>                                    env_tpe;
        typedef typename result_of::context<env_type, default_actions>::type      ctx_type
        typedef typename boost::result_of<evaluator(Expr const&, ctx_type)>::type type;
    };
}
```

Phoenix Expressions

A Phoenix Expression is a model of the Proto Expression Concept. These expressions are wrapped inside an Actor template. The `actor` provides the function call operator which evaluates the expressions. The `actor` is the domain specific wrapper around Phoenix expressions.

By design, Phoenix Expressions do not carry any information on how they will be evaluated later on. They are the data structure on which the `Actions` will work.

The library provides a convenience template to define expressions:

```
template <template <typename> Actor, typename Tag, typename A0, ..., typename A1>
struct expr_ext
    : proto::transform<expr_ext<Actor, Tag, A0, ..., A1> >
{
    typedef unspecified base_expr;
    typedef Actor<base_expr> type;

    typedef unspecified proto_grammar;

    static type make(A0 a0, ..., A1 a1);
};

template <typename Tag, typename A0, ..., typename A1>
struct expr : expr_ext<actor, Tag, A0, ..., A1> {};
```

Notation

`A0...AN` Child node types

`a0...aN` Child node objects

`G0...GN` Boost.Proto grammar types

Expression Semantics

Expression	Semantics
`expr<Tag, A0...AN>::type`	The type of Expression having tag `Tag` and `A0...AN` children
`expr<Tag, G0...GN>`	A Boost.Proto grammar and Proto Pass Through Transform
`expr<Tag, A0...AN>::make(a0...aN)`	Returns a Phoenix Expression

Note

You might have noticed the template template argument `Actor` used in `expr_ext`. This can be a user supplied custom Actor adding other member functions or objects than the default `actor` template. See Extending Actors for more details.

meta_grammar

Defining expressions is only part of the game to make it a valid Phoenix Expression. In order to use the expressions in the Phoenix domain, we need to "register" them to our grammar.

The `meta_grammar` is a struct for exactly that purpose. It is an openly extendable Boost.Proto Grammar:

```
struct meta_grammar
    : proto::switch_<meta_grammar>
{
    template <typename Tag, typename Dummy>
    struct case_
        : proto::not_<proto::_>
    {};
};
```

As you can see, by default the `meta_grammar` matches nothing. With every Module you include this grammar gets extended by various expressions.

Example

Define an expression:

```
template <typename Lhs, typename Rhs>
struct plus
    : expr<proto::tag::plus, Lhs, Rhs>
{};
```

And add it to the grammar:

```
template <>
struct meta_grammar::case_<proto::tag::plus>
    : enable_rule<
        plus<
            meta_grammar
          , meta_grammar
        >
    >
{};
```

Define a generator function to make the life of our potential users easier:

```
template <typename Lhs, typename Rhs>
typename plus<Lhs, Rhs>::type
plus(Lhs const & lhs, Rhs const & rhs)
{
    return expression::plus<Lhs, Rhs>::make(lhs, rhs);
}
```

Look if it really works:

```
plus(6, 5)();
```

returns 11!

```
proto::display_expr(plus(5, 6));
```

prints:

```
plus(
    terminal(6)
  , terminal(5)
)
```

See define_expression.cpp for the full example.

Note

The example shown here only works because `default_actions` knows how to handle an expression having the `proto::tag::plus` and two children. This is because `default_actions` uses the `proto::_default<meta_grammar>` transform to evaluate operators and functions. Learn more about actions here.

Boilerplate Macros

When having more and more expressions, you start to realize that this is a very repetetive task. Phoenix provides boilerplate macros that make defining Phoenix Expressions as you have seen in the previous section look like a piece of cake.

BOOST_PHOENIX_DEFINE_EXPRESSION

Description

`BOOST_PHOENIX_DEFINE_EXPRESSION` is a macro that can be used to generate all the necessary boilerplate to create Phoenix Expressions

Synopsis

```
BOOST_PHOENIX_DEFINE_EXPRESSION(
    (namespace_seq)(name)
  , (child_grammar0)
    (child_grammar1)
    ...
)
```

Semantics

The above macro generates the necessary code for an expression `name` in namespace `namespace_seq`. The sequence of `(child_grammarN)` declares how many children the expression will have and what `proto::grammar` they match.

The macro should be used at global scope. `namespace_seq` shall be the sequence of namespaces under which the following symbols will be defined:

```
namespace tag
{
    struct name;
}

namespace expression
{
    template <typename A0, typename A1 ... typename AN>
    struct name
        : boost::phoenix::expr<
            tag::name
          , A0
          , A1
            ...
          , AN
        >
}

namespace rule
{
    struct name
        : boost::phoenix::expr<
            child_grammar0
          , child_grammar1
            ...
          , child_grammarN
        >
    {};
}

namespace functional
{
    struct make_name; // A polymorphic function object that can be called to create the expres↵
sion node
}

namespace result_of
{
    template <typename A0, typename A1 ... typename AN>
    struct make_name; // The result type of the expression node
}

// convenience polymorphic function to create an expression node
template <typename A0, typename A1 ... typename AN>
result_of::make_name<A0, A1 ... AN>
make_name(A0 const & a0, A1 const & a1 ... AN const & an);
```

This macros also adds a specialization for `meta_grammar::case_<tag::name>` to enable the rule for further use in actions.

Header

```
#include <boost/phoenix/core/expression.hpp>
```

Example

The example from the previous section can be rewritten as:

```
BOOST_PHOENIX_DEFINE_EXPRESSION(
    (plus)
  , (meta_grammar)                // Lhs
    (meta_grammar)                // Rhs
)

template <typename Lhs, typename Rhs>
typename plus<Lhs, Rhs>::type
plus(Lhs const & lhs, Rhs const & rhs)
{
    return expression::plus<Lhs, Rhs>::make(lhs, rhs);
}
```

BOOST_PHOENIX_DEFINE_EXPRESSION_VARARG

Description

`BOOST_PHOENIX_DEFINE_EXPRESSION_VARARG` is a macro that can be used to generate all the necessary boilerplate to create Phoenix Expressions

Synopsis

```
BOOST_PHOENIX_DEFINE_EXPRESSION_VARARG(
    (namespace_seq)(name)
  , (child_grammar0)
    (child_grammar1)
    ...
    (child_grammarN)
  , N
)
```

Semantics

The above macro generates the necessary code for an expression `name` in namespace `namespace_seq`. `N` is the maximum number of variable children. All but the last elements in the grammar sequence are required children of the expression, and the last denotes a variable number of children. The number of children an expression of this kind can hold is therefor `N-1` plus the size of the sequence

The macro should be used at global scope. `namespace_seq` shall be the sequence of namespaces under which the following symbols will be defined:

```
namespace tag
{
    struct name;
}

namespace expression
{
    template <typename A0, typename A1 ... typename AN>
    struct name
        : boost::phoenix::expr<
            tag::name
          , A0
          , A1
            ...
          , AN
        >
    {};
}

namespace rule
{
    struct name
        : expression::name<
            child_grammar0
          , child_grammar1
            ...
          , proto::vararg<child_grammarN>
        >
    {};
}

namespace functional
{
    struct make_name; // A polymorphic function object that can be called to create the expres↵
sion node
}

namespace result_of
{
    template <typename A0, typename A1 ... typename AN>
    struct make_name; // The result type of the expression node
}

// convenience polymorphic function to create an expression node
template <typename A0, typename A1 ... typename AN>
result_of::make_name<A0, A1 ... AN>
make_name(A0 const & a0, A1 const & a1 ... AN const & an);
```

This macros also adds a specialization for `meta_grammar::case_<tag::name>` to enable the rule for further use in actions.

Header

```
#include <boost/phoenix/core/expression.hpp>
```

Example

```
BOOST_PHOENIX_DEFINE_EXPRESSION_VARARG(
    (boost)(phoenix)(mem_fun_ptr)
  , (meta_grammar)                // Pointer to Object
    (meta_grammar)                // Member pointer
    (meta_grammar)                // Variable number of arguments
  , BOOST_PHOENIX_LIMIT
)
```

This defines the member function pointer operator expression as described in operators.

BOOST_PHOENIX_DEFINE_EXPRESSION_EXT

Description

`BOOST_PHOENIX_DEFINE_EXPRESSION_EXT` is a macro that can be used to generate all the necessary boilerplate to create Phoenix Expressions

Synopsis

```
BOOST_PHOENIX_DEFINE_EXPRESSION_EXT(
    actor
  , (namespace_seq)(name)
  , (child_grammar0)
    (child_grammar1)
    ...
    (child_grammarN)
  , N
)
```

Semantics

The above macro generates the necessary code for an expression `name` in namespace `namespace_seq`. The sequence of `(child_grammarN)` declares how many children the expression will have and what `proto::grammar` they match.

The macro should be used at global scope. `namespace_seq` shall be the sequence of namespaces under which the following symbols will be defined:

```
namespace tag
{
    struct name;
}

namespace expression
{
    template <typename A0, typename A1 ... typename AN>
    struct name
        : boost::phoenix::expr_ext<
            actor
          , tag::name
          , A0
          , A1
            ...
          , AN
        >
}

namespace rule
{
    struct name
        : boost::phoenix::expr<
            child_grammar0
          , child_grammar1
            ...
          , child_grammarN
        >
    {};
}

namespace functional
{
    struct make_name; // A polymorphic function object that can be called to create the expres↵
sion node
}

namespace result_of
{
    template <typename A0, typename A1 ... typename AN>
    struct make_name; // The result type of the expression node
}

// convenience polymorphic function to create an expression node
template <typename A0, typename A1 ... typename AN>
result_of::make_name<A0, A1 ... AN>
make_name(A0 const & a0, A1 const & a1 ... AN const & an);
```

This macros also adds a specialization for `meta_grammar::case_<tag::name>` to enable the rule for further use in actions.

Header

```
#include <boost/phoenix/core/expression.hpp>
```

Example

```
BOOST_PHOENIX_DEFINE_EXPRESSION_EXT(
    if_actor
  , (boost)(phoenix)(if_)
  , (meta_grammar) // Cond
    (meta_grammar) // Then
)
```

This defines the if_ expression. The custom actor defines `else_` as a member.

BOOST_PHOENIX_DEFINE_EXPRESSION_EXT_VARARG

Description

`BOOST_PHOENIX_DEFINE_EXPRESSION_EXT_VARARG` is a macro that can be used to generate all the necessary boilerplate to create Phoenix Expressions

Synopsis

```
BOOST_PHOENIX_DEFINE_EXPRESSION_EXT_VARARG(
    actor
  , (namespace_seq)(name)
  , (child_grammar0)
    (child_grammar1)
    ...
    (child_grammarN)
  , N
)
```

Semantics

The above macro generates the necessary code for an expression `name` in namespace `namespace_seq`. `N` is the maximum number of variable children. All but the last elements in the grammar sequence are required children of the expression, and the last denotes a variable number of children. The number of children an expression of this kind can hold is therefor `N-1` plus the size of the sequence

The macro should be used at global scope. `namespace_seq` shall be the sequence of namespaces under which the following symbols will be defined:

```
namespace tag
{
    struct name;
}

namespace expression
{
    template <typename A0, typename A1 ... typename AN>
    struct name
        : boost::phoenix::expr_ext<
            actor
          , tag::name
          , A0
          , A1
            ...
          , AN
        >
    {};
}

namespace rule
{
    struct name
        : expression::name<
            child_grammar0
          , child_grammar1
            ...
          , proto::vararg<child_grammarN>
        >
    {};
}

namespace functional
{
    struct make_name; // A polymorphic function object that can be called to create the expres↵
sion node
}

namespace result_of
{
    template <typename A0, typename A1 ... typename AN>
    struct make_name; // The result type of the expression node
}

// convenience polymorphic function to create an expression node
template <typename A0, typename A1 ... typename AN>
result_of::make_name<A0, A1 ... AN>
make_name(A0 const & a0, A1 const & a1 ... AN const & an);
```

This macros also adds a specialization for `meta_grammar::case_<tag::name>` to enable the rule for further use in actions.

Header

```
#include <boost/phoenix/core/expression.hpp>
```

Example

TBD

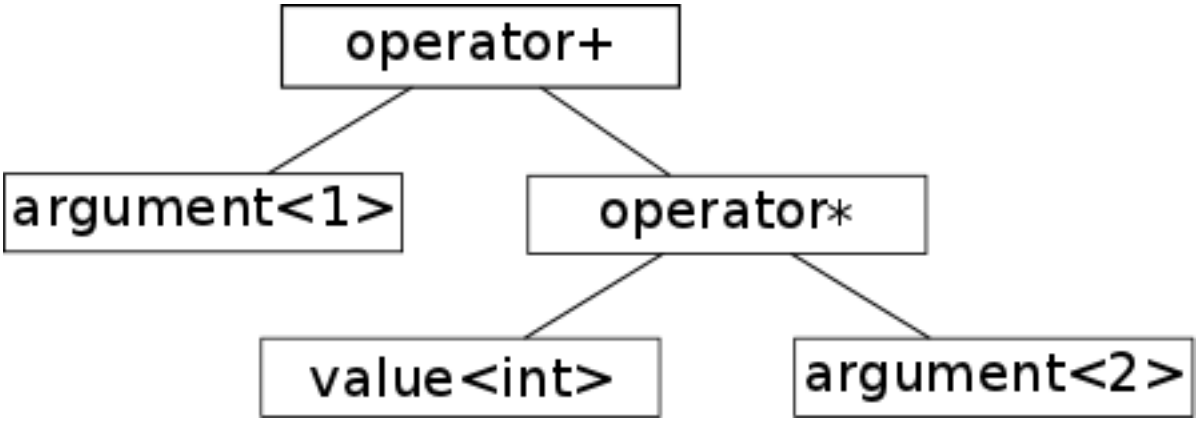
More on Actions

As you know from the Actors in Detail section, Actions are what brings life to a Phoenix expression tree.

When dealing with a Phoenix expression tree, it gets evaluated top-down. Example:

```
_1 + 3 * _2
```

Can be visualized as an AST in the following way:



In terms of actions this means:

- `rule::plus` is matched
- evaluate left:
  - `rule::placeholder` is matched
- evaluate right:
  - `rule::multiplies` is matched
    - evaluate left:
      - `rule::value` is matched
    - evaluate right:
      - `rule::placeholder` is matched

Every time a rule is matched, an action will be called. The action determines how the Phoenix AST will be traversed.

Writing an Action

As mentioned in Actors in Detail actions are Proto Primitive Transforms for convenience Phoenix provides an abstraction to this:

```
template <typename Fun>
struct call;
```

This is similar to `proto::call` but does more. It calls the `Fun` function object passed as template parameter with the `Context` and the children of the expression associated with the rule.

Lets have an (simplified) example on how to write an evaluation action for `rule::plus`:

```
struct plus_eval
{
    typedef int result_type;

    template <typename Lhs, typename Rhs, typename Context>
    result_type operator()(Lhs const& lhs, Rhs const &rhs, Context & ctx)
    {
        return eval(lhs, ctx) + eval(rhs, ctx);
    }
};

template <>
struct default_actions::when<rule::plus>
    : call<plus_eval>
{};
```

That's it. When evaluating a `plus` expression, the `plus_eval` callable gets called with the left hand side and right hand side expression and the associated Context.

But there is more: As Actions *can* be full fletched Proto Transforms, you can in fact use any proto expression you can imagine as the action. Phoenix predifines a set of callables and transform to deal with the Context information passed along and of course every Phoenix expression can be used as a Phoenix grammar or Proto Pass Through Transform.

`functional::context(Env, Actions)`	A Proto Callable Transform that creates a new context out of the `Env` and `Actions` parameter
`functional::env(Context)`	A Proto Callable Transform that returns the environment out of the `Context` parameter
`functional::actions(Context)`	A Proto Callable Transform that returns the actions out of the `Context` parameter
`_context`	A Proto Primitive Transform that returns the current context
`_env`	A Proto Primitive Transform that returns the current environment
`_actions`	A Proto Primitive Transform that returns the current actions
`context(env, actions)`	A regular function that creates a context
`env(ctx)`	A regular function that returns the environment from the given context
`actions(ctx)`	A regular function that returns the actions from the given context

Phoenix is equipped with a predefined set of expressions, rules and actions to make all the stuff work you learned in the Starter Kit and Modules sections. See the next section for more details!

Predefined Expressions and Rules

This section is the "behind the scenes" counter part of the Modules section. A listing of all the predefined expressions and rules:

Expression	Rule
`expression::value<T>`	`rule::value : expression::value<proto::_>`
`expression::reference<T>`	`rule::custom_terminal`
`expression::argument<N>`	`rule::argument`
`expression::null`	`rule::custom_terminal`
`expression::function<F, A0, ..., AN>`	`rule::function : expression::function<vararg<meta_grammar> >`
`expression::negate<A0>`	`rule::negate : expression::negate<meta_grammar>`
`expression::unary_plus<A0>`	`rule::negate : expression::unary_plus<meta_grammar>`
`expression::pre_inc<A0>`	`rule::negate : expression::pre_inc<meta_grammar>`
`expression::pre_dec<A0>`	`rule::negate : expression::pre_dec<meta_grammar>`
`expression::post_inc<A0>`	`rule::negate : expression::post_inc<meta_grammar>`
`expression::post_dec<A0>`	`rule::negate : expression::post_dec<meta_grammar>`
`expression::plus_assign<Lhs, Rhs>`	`rule::plus_assign : expression::plus_assign<meta_grammar, meta_grammar>`

Expression	Rule
expression::minus_assign<Lhs, Rhs>	rule::minus_assign : expression::minus_assign<meta_gram↵ mar, meta_grammar>
expression::multiplies_assign<Lhs, Rhs>	rule::multiplies_assign : expression::multiplies_as↵ sign<meta_grammar, meta_grammar>
expression::divides_assign<Lhs, Rhs>	rule::divides_assign : expression::divides_assign<meta_gram↵ mar, meta_grammar>
expression::modules_assign<Lhs, Rhs>	rule::modules_assign : expression::modules_assign<meta_gram↵ mar, meta_grammar>
expression::plus<Lhs, Rhs>	rule::plus : expression::plus<meta_gram↵ mar, meta_grammar>
expression::minus<Lhs, Rhs>	rule::minus : expression::minus<meta_gram↵ mar, meta_grammar>
expression::multiplies<Lhs, Rhs>	rule::multiplies : expression::multiplies<meta_gram↵ mar, meta_grammar>
expression::divides<Lhs, Rhs>	rule::divides : expression::divides<meta_gram↵ mar, meta_grammar>
expression::modulus<Lhs, Rhs>	rule::modulus : expression::modulus<meta_gram↵ mar, meta_grammar>
expression::complement<A0>	rule::complement : expression::complement<A0>

Expression	Rule
`expression::bitwise_and_assign<Lhs, Rhs>`	`rule::bitwise_and_assign : expression::bitwise_and_assign<meta_grammar, meta_grammar>`
`expression::bitwise_or_assign<Lhs, Rhs>`	`rule::bitwise_or_assign : expression::bitwise_or_assign<meta_grammar, meta_grammar>`
`expression::bitwise_xor_assign<Lhs, Rhs>`	`rule::bitwise_xor_assign : expression::bitwise_xor_assign<meta_grammar, meta_grammar>`
`expression::shift_left_assign<Lhs, Rhs>`	`rule::shift_left_assign : expression::shift_left_assign<meta_grammar, meta_grammar>`
`expression::shift_right_assign<Lhs, Rhs>`	`rule::shift_right_assign : expression::shift_right_assign<meta_grammar, meta_grammar>`
`expression::bitwise_and<Lhs, Rhs>`	`rule::bitwise_and : expression::bitwise_and<meta_grammar, meta_grammar>`
`expression::bitwise_or<Lhs, Rhs>`	`rule::bitwise_or : expression::bitwise_or<meta_grammar, meta_grammar>`
`expression::bitwise_xor<Lhs, Rhs>`	`rule::bitwise_xor : expression::bitwise_xor<meta_grammar, meta_grammar>`
`expression::shift_left<Lhs, Rhs>`	`rule::shift_left : expression::shift_left<meta_grammar, meta_grammar>`
`expression::shift_right<Lhs, Rhs>`	`rule::shift_right : expression::shift_right<meta_grammar, meta_grammar>`

Expression	Rule
`expression::equal_to<Lhs, Rhs>`	`rule::equal_to : expression::equal_to<meta_grammar, meta_grammar>`
`expression::not_equal_to<Lhs, Rhs>`	`rule::not_equal_to : expression::not_equal_to<meta_grammar, meta_grammar>`
`expression::less_equal_to<Lhs, Rhs>`	`rule::less_equal_to : expression::less_equal_to<meta_grammar, meta_grammar>`
`expression::greater_equal<Lhs, Rhs>`	`rule::greater_equal : expression::greater_equal<meta_grammar, meta_grammar>`
`expression::less<Lhs, Rhs>`	`rule::less : expression::less<meta_grammar, meta_grammar>`
`expression::greater<Lhs, Rhs>`	`rule::greater : expression::greater<meta_grammar, meta_grammar>`
`expression::if_else_operator<Cond, Then, Else>`	`rule::if_else : expression::if_else<meta_grammar, meta_grammar, meta_grammar>`
`expression::logical_not<A0>`	`rule::logical_not : expression::logical_not<meta_grammar>`
`expression::logical_and<Lhs, Rhs>`	`rule::logical_and : expression::logical_and<meta_grammar, meta_grammar>`
`expression::logical_or<Lhs, Rhs>`	`rule::logical_or : expression::logical_or<meta_grammar, meta_grammar>`

Expression	Rule
`expression::mem_fun_ptr<Object, MemPtr, A0, ..., AN>`	`rule::mem_fun_ptr : expression::mem_fun_ptr<meta_grammar, meta_grammar, vararg<meta_grammar> >`
`expression::address_of<A0>`	`rule::address_of : expression::address_of<meta_grammar>`
`expression::dereference<A0>`	`rule::dereference : expression::dereference<meta_grammar>`
`expression::assign<Lhs, Rhs>`	`rule::assign : expression::assign<meta_grammar, meta_grammar>`
`expression::subscript<Lhs, Rhs>`	`rule::subscript : expression::subscript<meta_grammar, meta_grammar>`
`expression::sequence<A0, A1>`	`rule::sequence : expression::sequence<meta_grammar, meta_grammar>`
`expression::if_<Cond, Then>`	`rule::if_ : expression::if_<meta_grammar, meta_grammar>`
`expression::if_else_statement<Cond, Then, Else>`	`rule::if_else_statement : expression::if_else_statement<meta_grammar, meta_grammar, meta_grammar>`
`expression::switch_case<Label, Statement>`	`rule::switch_case : expression::switch_case< terminal<mpl::int_<N> >, meta_grammar>`
`expression::switch_default_case<Statement>`	`rule::switch_default_case : expression::switch_default_case<meta_grammar>`

Expression	Rule
`expression::switch_<Cond, Cases>`	`rule::switch_ : expression::switch_< meta_grammar , switch_grammar >`
`expression::while_<Cond, Do>`	`rule::while_ : expression::while_< meta_grammar , meta_grammar >`
`expression::do_while<Cond, Do>`	`rule::do_while : expression::do_while< meta_grammar , meta_grammar >`
`expression::for_<Init, Cond, Step, Do>`	`rule::for_ : expression::for_< meta_grammar , meta_grammar , meta_grammar , meta_grammar >`
`expression::catch_<Exception, Statement>`	`rule::catch_ : expression::catch_< catch_exception<proto::_> , meta_grammar >`
`expression::catch_all<Statement>`	`rule::catch_all : expression::catch_<meta_grammar>`

Expression	Rule
`expression::try_catch<Try, Catch0, ..., CatchN>`	`rule::try_catch : proto::or_< expression::try_catch< meta_grammar , vararg<rule::catch_> > , expression::try_catch< meta_grammar , vararg<rule::catch_> , rule::catch_all > , expression::try_catch< meta_grammar , catch_all > >`
`expression::throw_<A0>`	`rule::throw_ : expression::throw_<meta_grammar>`
`expression::construct<Target, A0, ..., AN>`	`rule::construct : expression::construct< terminal<detail::target<proto::_> > , A0 ... , AN >`
`expression::new_<Target, A0, ..., AN>`	`rule::new_ : expression::new_< terminal<detail::target<proto::_> > , A0 ... , AN >`
`expression::delete_<A0>`	`rule::delete_ : expression::delete_<meta_grammar>`
`expression::static_cast_<Target, A>`	`rule::static_cast_ : expression::static_cast_< terminal<detail::target<proto::_> > , A >`

Expression	Rule
`expression::dynamic_cast_<Target, A>`	`rule::dynamic_cast_ : expression::dynamic_cast_< terminal<detail::target<proto::_> > , A >`
`expression::reinterpret_cast_<Target, A>`	`rule::reinterpret_cast_ : expression::reinterpret_cast_< terminal<detail::target<proto::_> > , A >`
`expression::const_cast_<Target, A>`	`rule::const_cast_ : expression::const_cast_< terminal<detail::target<proto::_> > , A >`
`expression::local_variable<Key>`	`rule::custom_terminal`
`expression::let<Locals, Statement`	`rule::let : expression::let<termin↵al<proto::_>, meta_grammar>`
`expression::lambda<OuterEnv, Locals, Statement`	`rule::lambda : expression::lambda<termin↵al<proto::_>, terminal<proto::_>, meta_gram↵mar>`
`expression::lambda_actor<Locals, Statement`	`rule::lambda_actor : expression::lambda_actor<termin↵al<proto::_>, meta_grammar>`

Custom Terminals

Custom Terminals are used in Phoenix to handle special values transparently. For example, as Phoenix captures everything by value, we needed to use `boost::reference_wrapper` to bring reference semantics into Phoenix.

Custom terminals could be any wrapper class:

```
template <typename T>
struct is_custom_terminal;
```

needs to be specialized in order for Phoenix to recognize this wrapper type. `default_action` calls `custom_terminal<T>`.

Example:

```
// Call out boost::reference_wrapper for special handling
template<typename T>
struct is_custom_terminal<boost::reference_wrapper<T> >
  : mpl::true_
{};

// Special handling for boost::reference_wrapper
template<typename T>
struct custom_terminal<boost::reference_wrapper<T> >
{
    typedef T &result_type;

    template <typename Context>
    T &operator()(boost::reference_wrapper<T> r, Context &) const
    {
        return r;
    }
};
```

Placeholder Unification

Phoenix uses `boost::is_placeholder` for recognizing placeholders:

```
template <typename T>
struct is_placeholder
{
    static const int value = 0;
};
```

To adapt your own placeholder, the nested value needs to be greater than 0 for your types. This is done by specializing this trait.

Advanced Examples

Extending Actors

Actors are one of the main parts of the library, and one of the many customization points. The default actor implementation provides several operator() overloads which deal with the evaluation of expressions.

For some use cases this might not be enough. For convenience it is thinkable to provide custom member functions which generate new expressions. An example is the if_else_ Statement which provides an additional else member for generating a lazy if-else expression. With this the actual Phoenix expression becomes more expressive.

Another scenario is to give actors the semantics of a certain well known interface or concept. This tutorial like section will provide information on how to implement a custom actor which is usable as if it were a STL Container.

Requirements

Let's repeat what we want to have:

Expression	Semantics
`a.begin()`	Returns an iterator pointing to the first element in the container.
`a.end()`	Returns an iterator pointing one past the last element in the container.
`a.size()`	Returns the size of the container, that is, its number of elements.
`a.max_size()`	Returns the largest size that this container can ever have.
`a.empty()`	Equivalent to a.size() == 0. (But possibly faster.)
`a.swap(b)`	Equivalent to swap(a,b)

Additionally, we want all the operator() overloads of the regular actor.

Defining the actor

The first version of our `container_actor` interface will show the general principle. This will be continually extended. For the sake of simplicity, every member function generator will return `nothing` at first.

```
template <typename Expr>
struct container_actor
 : actor<Expr>
{
 typedef actor<Expr> base_type;
 typedef container_actor<Expr> that_type;

 container_actor( base_type const& base )
  : base_type( base ) {}

 expression::null<mpl::void_>::type const begin() const { return nothing; }
 expression::null<mpl::void_>::type const end() const { return nothing; }
 expression::null<mpl::void_>::type const size() const { return nothing; }
 expression::null<mpl::void_>::type const max_size() const { return nothing; }
 expression::null<mpl::void_>::type const empty() const { return nothing; }

 // Note that swap is the only function needing another container.
 template <typename Container>
 expression::null<mpl::void_>::type const swap( actor<Container> const& ) const { return nothing; }
};
```

Using the actor

Although the member functions do nothing right now, we want to test if we can use our new actor.

First, lets create a generator which wraps the `container_actor` around any other expression:

```
template <typename Expr>
container_actor<Expr> const
container( actor<Expr> const& expr )
{
    return expr;
}
```

Now let's test this:

```
std::vector<int> v;
v.push_back(0);
v.push_back(1);
v.push_back(2);
v.push_back(3);

(container(arg1).size())(v);
```

Granted, this is not really elegant and not very practical (we could have just used phoenix::begin(v) from the Phoenix algorithm module, but we can do better.

Let's have an argument placeholder which is usable as if it was a STL container:

```
container_actor<expression::argument<1>::type> const con1;
// and so on ...
```

The above example can be rewritten as:

```
std::vector<int> v;
v.push_back(0);
v.push_back(1);
v.push_back(2);
v.push_back(3);

(con1.size())(v);
```

Wow, that was easy!

Adding life to the actor

This one will be even easier!

First, we define a lazy function which evaluates the expression we want to implement. Following is the implementation of the size function:

```
struct size_impl
{
 // result_of protocol:
 template <typename Sig>
 struct result;

 template <typename This, typename Container>
 struct result<This(Container)>
 {
  // Note, remove reference here, because Container can be anything
  typedef typename boost::remove_reference<Container>::type container_type;

  // The result will be size_type
  typedef typename container_type::size_type type;
 };

 template <typename Container>
 typename result<size_impl(Container const&)>::type
 operator()(Container const& container) const
 {
  return container.size();
 }
};
```

Good, this was the first part. The second part will be to implement the size member function of `container_actor`:

```
template <typename Expr>
struct container_actor
 : actor<Expr>
{
 typedef actor<Expr> base_type;
 typedef container_actor<Expr> that_type;

 container_actor( base_type const& base )
  : base_type( base ) {}

 typename expression::function<size_impl, that_type>::type const
 size() const
 {
  function<size_impl> const f = size_impl();
  return f(*this);
 }

 // the rest ...
};
```

It is left as an exercise to the user to implement the missing parts by reusing functions from the Phoenix Algorithm Module (the impatient take a look here: container_actor.cpp).

Adding an expression

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 eill present it here in its entirety (verbatim):

```
BOOST_PHOENIX_DEFINE_EXPRESSION(
    (boost)(phoenix)(while_)
  , (meta_grammar)          // Cond
    (meta_grammar)          // Do
)

namespace boost { namespace phoenix
{
    struct while_eval
    {
        typedef void result_type;

        template <typename Cond, typename Do, typename Context>
        result_type
        operator()(Cond const& cond, Do const& do_, Context & ctx) const
        {
            while(eval(cond, ctx))
            {
                eval(do_, ctx);
            }
        }
    };

    template <typename Dummy>
    struct default_actions::when<rule::while_, Dummy>
        : call<while_eval, Dummy>
    {};

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

        template <typename Do>
        typename expression::while_<Cond, Do>::type const
        operator[](Do const& do_) const
        {
            return expression::while_<Cond, Do>::make(cond, do_);
        }

        Cond const& cond;
    };

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

`while_eval` is an example of how to evaluate an expression. It gets called in the `rule::while` action. `while_gen` and `while_` are the expression template front ends. Let's break this apart to undestand 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:

```
expression::while_<Cond, Do>::type
```

where `Cond` is the type of `cond` and `Do` is the type of `do_`. Notice how we are using Phoenix's Expression mechanism here

```
template <typename Do>
typename expression::while_<Cond, Do>::type const
operator[](Do const& do_) const
{
    return expression::while_<Cond, Do>::make(cond, do_);
}
```

Finally, the `while_eval` does its thing:

```
while(eval(cond, ctx))
{
    eval(do_, ctx);
}
```

`cond` and `do_`, at this point, are instances of Actor. `cond` and `do_` are the Actors passed as parameters by `call`, ctx is the Context

Transforming the Expression Tree

This example will show how to write Actions that transform the Phoenix AST.

> "*Lisp macros transform the program structure itself, with the full language available to express such transformations.*"
>
> Wikipedia

What we want to do is to invert some arithmetic operators, i.e. plus will be transformed to minus, minus to plus, multiplication to division and division to multiplication.

Let's start with defining our default action:

```
struct invert_actions
{
    template <typename Rule>
    struct when
        : proto::_ // the default is proto::_
    {};
};
```

By default, we don't want to do anything, well, not exactly nothing, but just return the expression. This is done by proto::_ which, used as a transform, just passes the current expression along. Making this action an identity transform.

So, after the basics are set up, we can start by writing the transformations we want to have on our tree:

```
// Transform plus to minus
template <>
struct invert_actions::when<phoenix::rule::plus>
    : proto::call<
        proto::functional::make_expr<proto::tag::minus>(
            phoenix::evaluator(proto::_left, phoenix::_context)
          , phoenix::evaluator(proto::_right, phoenix::_context)
        )
    >
{};
```

Wow, this looks complicated! Granted you need to know a little bit about Boost.Proto (For a good introduction read through the Expressive C++ series).

What is done is the following:

- The left expression is passed to evaluator (with the current context, that contains our invert_actions)
- The right expression is passed to evaluator (with the current context, that contains our invert_actions)
- The result of these two Proto Transforms is passed to `proto::functional::make_expr` which returns the freshly created expression

After you know what is going on, maybe the rest doesn't look so scary anymore:

```
// Transform minus to plus
template <>
struct invert_actions::when<phoenix::rule::minus>
    : proto::call<
        proto::functional::make_expr<proto::tag::plus>(
            phoenix::evaluator(proto::_left, phoenix::_context)
          , phoenix::evaluator(proto::_right, phoenix::_context)
        )
    >
{};

// Transform multiplies to divides
template <>
struct invert_actions::when<phoenix::rule::multiplies>
    : proto::call<
        proto::functional::make_expr<proto::tag::divides>(
            phoenix::evaluator(proto::_left, phoenix::_context)
          , phoenix::evaluator(proto::_right, phoenix::_context)
        )
    >
{};

// Transform divides to multiplies
template <>
struct invert_actions::when<phoenix::rule::divides>
    : proto::call<
        proto::functional::make_expr<proto::tag::multiplies>(
            phoenix::evaluator(proto::_left, phoenix::_context)
          , phoenix::evaluator(proto::_right, phoenix::_context)
        )
    >
{};
```

That's it! Now that we have our actions defined, we want to evaluate some of our expressions with them:

```
template <typename Expr>
// Calculate the result type: our transformed AST
typename boost::result_of<
    phoenix::evaluator(
        Expr const&
      , phoenix::result_of::context<int, invert_actions>::type
    )
>::type
invert(Expr const & expr)
{
    return
        // Evaluate it with our actions
        phoenix::eval(
            expr
          , phoenix::context(
                int()
              , invert_actions()
            )
        );
}
```

Run some tests to see if it is working:

```
invert(_1);                   // --> _1
invert(_1 + _2);              // --> _1 - _2
invert(_1 + _2 - _3);         // --> _1 - _2 + _3
invert(_1 * _2);              // --> _1 / _2
invert(_1 * _2 / _3);         // --> _1 / _2 * _3
invert(_1 * _2 + _3);         // --> _1 / _2 - _3
invert(_1 * _2 - _3);         // --> _1 / _2 + _2
invert(if_(_1 * _4)[_2 - _3]); // --> if_(_1 / _4)[_2 + _3]
_1 * invert(_2 - _3));        // --> _1 * _2 + _3
```


The complete example can be found here: example/invert.cpp

Pretty simple ...

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 millennium.

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.

Acknowledgments

1. Hartmut Kaiser implemented the original lazy casts and constructors based on his original work on Spirit SE "semantic expressions" (the precursor to Phoenix), and guided Phoenix from the initial review of V2 to the release of V3.

2. Eric Niebler did a 2.0 pre-release review and wrote some range related code that Phoenix stole and used in the algorithms. Additionally he played the leading role in inventing the extension mechanism as well as providing early prototypes and helping with Boost.Proto code. DA Proto MAN!

3. Angus Leeming implemented the container functions on Phoenix-1 which I then ported to Phoenix-2.

4. 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.

5. Jaakko Jarvi. DA Lambda MAN!

6. Dave Abrahams, for his constant presence, wherever, whenever.

7. Aleksey Gurtovoy, DA MPL MAN!

8. Doug Gregor, always a source of inspiration.

9. Dan Marsden, did almost all the work in bringing Phoenix-2 out the door.

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/whyfp.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/~jajarvi/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 Jhonson, 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.