Author's Corner: Boost.Parameter

David Abrahams

BoostCon'08

Outline

- **What’s the Point?**
- Free Functions
- Member Functions
- Constructors & ArgumentPacks
- Class Templates
- Advanced Topics
  - Building ArgumentPacks
  - Extracting Parameter Types
  - Lazy Defaults
- Best Practices
- Python Binding

Getting Positional

```
window* new_window(
   char const* name,
   int border_width = default_border_width,
   bool movable = true,
   bool initially_visible = true
   );

const bool movability = false;
window* w = new_window("alert box", movability);

window* w = new_window("alert", 1, true, false);
```

With Boost.Parameter

- Named Function Parameters

```
window* w = new_window("alert box", _movable=false);
```

- Named Template Parameters

```
smart_ptr<ownership<shared>, value_type<Client> > p;
```

- Deduced Function Parameters

```
window* w = new_window(_movable=false, "alert box");
```

- Deduced Template Parameters

```
smart_ptr<shared, Client> p;
smart_ptr<Client, shared> q;
```

Outline

- What’s the Point?
- **Free Functions**
- Member Functions
- Constructors & ArgumentPacks
- Class Templates
- Advanced Topics
  - □ Building ArgumentPacks
  - □ Extracting Parameter Types
  - □ Lazy Defaults
- Best Practices
- Python Binding

A Real Example – “Ideal” Syntax

```
template <
    class Graph, class DFSVisitor, class Index, class ColorMap
>
void depth_first_search(
    Graph const& graph,   // Required

    DFSVisitor visitor = boost::dfs_visitor<>(),

    typename graph_traits<g>::vertex_descriptor
      root_vertex
        = *vertices(graph).first,

    IndexMap index_map
        = get(boost::vertex_index,graph),

    ColorMap& color_map
        = default_color_map(num_vertices(graph), index_map)
)
{ … }
```

A Real Example – Legal Syntax

```
template <
    class Graph, class DFSVisitor, class Index, class ColorMap
>
void depth_first_search(
    Graph const& graph,   // Required

    DFSVisitor visitor,

    typename graph_traits<g>::vertex_descriptor
      root_vertex,


    IndexMap index_map,


    ColorMap& color_map

)
{ … }
```

Declaring Keywords

```
namespace graphs
{
  BOOST_PARAMETER_NAME(graph)
  BOOST_PARAMETER_NAME(visitor)
  BOOST_PARAMETER_NAME(root_vertex)
  BOOST_PARAMETER_NAME(index_map)
  BOOST_PARAMETER_NAME(color_map)
}
```

Ideal Syntax Again

```
template <
    class Graph, class DFSVisitor, class Index, class ColorMap
>
 void  depth_first_search(
    Graph const& graph,   // Required

    DFSVisitor visitor = boost::dfs_visitor<>(),

    typename graph_traits<g>::vertex_descriptor
      root_vertex
        = *vertices(graph).first,

    IndexMap index_map
        = get(boost::vertex_index,graph),

    ColorMap& color_map
        = default_color_map(num_vertices(graph), index_map)
)
{ … }
```

Actual Declaration

```
namespace graphs {
BOOST_PARAMETER_FUNCTION(

(void),depth_first_search, tag,
    (required   (graph, *) )
    (optional
              (visitor,*,boost::dfs_visitor<>())


     (root_vertex, *,
          *vertices(graph).first)

             (index_map, *,
          get(boost::vertex_index,graph))

      (in_out(color_map), *,
          default_color_map(num_vertices(graph), index_map) )
))
{ … }
}
```

Actual Declaration – Re-Indented

```
namespace graphs {
BOOST_PARAMETER_FUNCTION(

(void),depth_first_search, tag,
    (required   (graph, *) )
    (optional
      (visitor,           *,
          boost::dfs_visitor<>())

      (root_vertex,       *,
          *vertices(graph).first)

      (index_map,         *,
          get(boost::vertex_index,graph))

      (in_out(color_map), *,
          default_color_map(num_vertices(graph), index_map) )
))
{ … }
}
```

Adding Type Requirements

```
BOOST_PARAMETER_FUNCTION(

(void),depth_first_search, tag,
    (required   (graph, *) )
    (optional
      (visitor,           *,
          boost::dfs_visitor<>())

      (root_vertex, (typename boost::graph_traits<
                              graph_type>::vertex_descriptor),

          *vertices(graph).first)
      (index_map,         *,
          get(boost::vertex_index,graph))

      (in_out(color_map), *,
          default_color_map(num_vertices(graph), index_map) )
))
{ … }
```

Adding Type Predicates

```
BOOST_PARAMETER_FUNCTION(

(void),depth_first_search, tag,
    (required   (graph, *) )
    (optional
      (visitor,           *,
          boost::dfs_visitor<>())

      (root_vertex, *(boost::is_convertible<
                          _, typename boost::graph_traits<
                              graph_type>::vertex_descriptor>),
          *vertices(graph).first)
      (index_map,         *,
          get(boost::vertex_index,graph))
      (in_out(color_map), *,
          default_color_map(num_vertices(graph), index_map) )
))
{ … }
```

Deduced Parameters – Ideal Syntax?

```
template <
  class Function, Class KeywordExpression,
  class CallPolicies
>
void def(
    char const* name, Callable func,

    char const* docstring = "",
    KeywordExpression keywords = no_keywords(),
    CallPolicies policies = default_call_policies()
);

// C++ Interface
Bar* f(Foo*);
double sin(double x = 3.14);

// Python Binding
def("f", &f, return_internal_reference<1>());
def("sin", &sin,
     "Takes the sine of its argument", (arg("x") = 3.14));
```

Actual Declaration

```
BOOST_PARAMETER_FUNCTION(
    (void), def, tag,

    (required (name,(char const*)) (func,*) )   // nondeduced

    (deduced
      (optional
        (docstring, (char const*), "")

        (keywords
           , *(is_keyword_expression<mpl::_>)
           , no_keywords())

        (policies
           , *(mpl::not_<
                 mpl::or_<
                     boost::is_convertible<mpl::_, char const*>
                   , is_keyword_expression<mpl::_>
                 >
             >)
           , default_call_policies()
         )
       )))
```

Syntax Summary

- Function Declaration

```
BOOST_PARAMETER_FUNCTION(
     (return type) ,  function name ,  keyword namespace ,

     (required ...parameters...   )
     (optional   ...parameters... )
     (deduced
        (required ...parameters... )
        (optional   ...parameters...)
     )
)
```

omit at most 2

omit at most 1

- Parameter Declaration

```
(name ,  type requirements ,  default value)
```

Outline

- What’s the Point?
- Free Functions
- **Member Functions**
- Constructors & ArgumentPacks
- Class Templates
- Advanced Topics
  - □ Building ArgumentPacks
  - □ Extracting Parameter Types
  - □ Lazy Defaults
- Best Practices
- Python Binding

Member Function Support

```
namespace test
{
  BOOST_PARAMETER_NAME(arg1)
  BOOST_PARAMETER_NAME(arg2)

  struct callable_with_2_ints
  {
      BOOST_PARAMETER_CONST_MEMBER_FUNCTION(
          (void), operator(), tag, (required (arg1,(int))(arg2,(int))))
      {
          std::cout << arg1 << ", " << arg2 << std::endl;
      }
  };
}

int main()
{
    test::callable_with_2_ints f;
    using namespace test::tag;

    f( _arg2=3, _arg1=5 );
}
```

Digression: Separate Compilation

```
struct callable_with_2_ints
{
    BOOST_PARAMETER_CONST_MEMBER_FUNCTION(
        (void), operator(), tag, (required (arg1,(int))(arg2,(int))))
    {
        call_impl(arg1, arg2);
    }

 private:
    void call_impl(int, int); // implemented elsewhere.
};
```

Outline

- What’s the Point?
- Free Functions
- Member Functions
- **Constructors & ArgumentPacks**
- Class Templates
- Advanced Topics
  - Building ArgumentPacks
  - Extracting Parameter Types
  - Lazy Defaults
- Best Practices
- Python Binding

Constructors & ArgumentPacks

```
BOOST_PARAMETER_NAME(name)
BOOST_PARAMETER_NAME(index)

struct myclass_impl
{
    template <class ArgumentPack>
    myclass_impl(ArgumentPack const& args)
    {
        std::cout << "name = " << args[_name]
                  << "; index = " << args[_index | 42]
                  << std::endl;
    }
};

struct myclass: myclass_impl
{
    BOOST_PARAMETER_CONSTRUCTOR(
        myclass, (myclass_impl), tag
      , (required (name,*)) (optional (index,*))) // no semicolon
};
```

Outline

- What’s the Point?
- Free Functions
- Member Functions
- Constructors & ArgumentPacks
- **Class Templates**
- Advanced Topics
  - Building ArgumentPacks
  - Extracting Parameter Types
  - Lazy Defaults
- Best Practices
- Python Binding

Template Parameters – Ideal Syntax

```
template <
    class class_type, class base_list = bases<>
  , class held_type = class_type, class copyable = void
>
class class_;

// C++ Interface
struct Var { char const* name; int value; };

struct PrintableVar : Var
{
    void print() { … };
};

// Python Bindings
class_<Var>("Var", init<std::string>())
    .def_readonly("name", &Var::name)
    .def_readwrite("value", &Var::value);


class_<PrintableVar, bases<Var> >("PrintableVar")
    .def("print", &PrintableVar::print);
```

Declaring Template Keywords

```
namespace boost { namespace python {

BOOST_PARAMETER_TEMPLATE_KEYWORD(class_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(base_list)
BOOST_PARAMETER_TEMPLATE_KEYWORD(held_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(copyable)

}}
```

```
namespace boost { namespace python {

namespace tag { struct base_list; } // keyword tag type

template <class T>
struct base_list
  : parameter::template_keyword<tag::base_list,T>
{};

}}
```

Declare Class Template Signature

```
namespace boost { namespace python {
using boost::mpl::_;

typedef parameter::parameters<
  required<tag::class_type, is_class<_> >,
  optional<tag::base_list, mpl::is_sequence<_> >,
  optional<tag::held_type>,
  optional<tag::copyable>
>
class_signature;

}}
```

```
template <
  class class_type,
  class base_list=bases<>,
  class held_type=class_type,
  class copyable=void
>
class class_;
```

Actual Class Template Declaration

```
namespace boost { namespace python {
using boost::mpl::_;

template <
  class A0,
  class A1 = parameter::void_,
  class A2 = parameter::void_,
  class A3 = parameter::void_
>
struct class_
{
    …
};

}}
```

```
template <
  class class_type,
  class base_list=bases<>,
  class held_type=class_type,
  class copyable=void
>
class class_;
```

Actual Class Template Declaration

```
template <
  class A0
  class A1 = parameter::void_,
  class A2 = parameter::void_,
  class A3 = parameter::void_
>
struct class_
{
    typedef typename
      class_signature::bind<A0,A1,A2,A3>::type
    args;

    typedef typename parameter::binding<
      args, tag::class_type>::type class_type;

    typedef typename parameter::binding<
      args, tag::base_list, bases<> >::type base_list;

    typedef typename parameter::binding<
      args, tag::held_type, class_type>::type held_type;

    typedef typename parameter::binding<
      args, tag::copyable, void>::type copyable;
};
```

Named Template Parameters: Usage

```
class B {};
class D : public B {};

typedef boost::python::class_<
    class_type<B>, copyable<boost::noncopyable>
> c1;

typedef boost::python::class_<
    D, held_type<std::auto_ptr<D> >, base_list<bases<B> >
> c2;

BOOST_MPL_ASSERT((boost::is_same<c1::class_type, B>));

BOOST_MPL_ASSERT((boost::is_same<c1::base_list, bases<> >));

BOOST_MPL_ASSERT((boost::is_same<c1::held_type, B>));

BOOST_MPL_ASSERT((
     boost::is_same<c1::copyable, boost::noncopyable>
));
```

Deduced Template Parameters

```
typedef parameter::parameters<
  required<tag::class_type, is_class<_> >,
  optional<tag::base_list, mpl::is_sequence<_> >,
  optional<tag::held_type>,
  optional<tag::copyable>
>
class_signature;
```

Deduced Template Parameters

```
typedef parameter::parameters<
  required<tag::class_type, is_class<_> >,
  optional<deduced<tag::base_list>, mpl::is_sequence<_> >,
  optional<tag::held_type>,
  optional<tag::copyable>
>
class_signature;
```

Deduced Template Parameters

```
typedef parameter::parameters<
  required<tag::class_type, is_class<_> >,
  optional<deduced<tag::base_list>, mpl::is_sequence<_> >,
  optional<deduced<tag::held_type> >,
  optional<tag::copyable>
>
class_signature;
```

Deduced Template Parameters

```
typedef parameter::parameters<
  required<tag::class_type, is_class<_> >,
  optional<deduced<tag::base_list>, mpl::is_sequence<_> >,
  optional<deduced<tag::held_type>, mpl::not_<mpl::is_sequence<_> > >,
  optional<tag::copyable>
>
class_signature;
```

Deduced Template Parameters

```
typedef parameter::parameters<
  required<tag::class_type, is_class<_> >,
  optional<deduced<tag::base_list>, mpl::is_sequence<_> >,
  optional<deduced<tag::held_type>, mpl::not_<mpl::is_sequence<_> > >,
  optional<deduced<tag::copyable>, boost::is_same<noncopyable,_> >
>
class_signature;
```

Deduced Template Parameters

```
typedef parameter::parameters<
  required<tag::class_type, is_class<_> >,
  optional<deduced<tag::base_list>, mpl::is_sequence<_> >,
  optional<
    deduced<tag::held_type>,
    mpl::and_<
      mpl::not_<mpl::is_sequence<_> > >,
      mpl::not_<boost::is_same<noncopyable,_> >
    > >,
  optional<deduced<tag::copyable>, boost::is_same<noncopyable,_> >
>
class_signature;


typedef boost::python::class_<B, boost::noncopyable> c1;

typedef boost::python::class_<D, std::auto_ptr<D>, bases<B> > c2;
```

Outline

- What’s the Point?
- Free Functions
- Member Functions
- Constructors & ArgumentPacks
- Class Templates
- **Advanced Topics**
  - □ Building ArgumentPacks
  - □ Extracting Parameter Types
  - □ Lazy Defaults
- Best Practices
- Python Binding

Building ArgumentPacks

```
BOOST_PARAMETER_NAME(index)

template <class ArgumentPack>
int print_index(ArgumentPack const& args)
{
    std::cout << "index = " << args[_index] << std::endl;
    return 0;
}
int x = print_index(_index = 3);

BOOST_PARAMETER_NAME(name)

template <class ArgumentPack>
int print_name_and_index(ArgumentPack const& args)
{
    std::cout << "name = " << args[_name] << "; ";
    return print_index(args);
}

int y = print_name_and_index((_index = 3, _name = "jones"));
```

Getting Positional Again

```
parameter::parameters<
    required<tag::name, is_convertible<_,char const*> >
  , optional<tag::index, is_convertible<_,int> >
> spec;

char const sam[] = "sam";
int twelve = 12;

int z0 = print_name_and_index( spec(sam, twelve) );

int z1 = print_name_and_index(
   spec(_index=12, _name="sam")
);
```

Deducing Argument Types

```
BOOST_PARAMETER_NAME(name)
BOOST_PARAMETER_NAME(index)

template <class Name, class Index>
int deduce_arg_types_impl(Name& name, Index& index)
{
    Name& n2 = name;  // we know the types
    Index& i2 = index;
    return index;
}

template <class ArgumentPack>
int deduce_arg_types(ArgumentPack const& args)
{
    return deduce_arg_types_impl(args[_name], args[_index|42]);
}
```

Extracting Argument Types

```
BOOST_PARAMETER_NAME(index)

template <class ArgumentPack>
typename remove_reference<
    typename parameter::binding<ArgumentPack, tag::index, int>::type
>::type
twice_index(ArgumentPack const& args)
{
    return 2 * args[_index|42];
}

int six = twice_index(_index = 3);
```

Extracting Argument Types

```
BOOST_PARAMETER_NAME(index)

template <class ArgumentPack>

    typename parameter::value_type<ArgumentPack, tag::index, int>::type

twice_index(ArgumentPack const& args)
{
    return 2 * args[_index|42];
}

int six = twice_index(_index = 3);
```

Lazy Defaults

```
BOOST_PARAMETER_NAME(s1)
BOOST_PARAMETER_NAME(s2)
BOOST_PARAMETER_NAME(s3)

template <class ArgumentPack>
std::string f(ArgumentPack const& args)
{
    std::string const& s1 = args[_s1];
    std::string const& s2 = args[_s2];
    typename parameter::binding<
        ArgumentPack,tag::s3,std::string
    >::type
      s3 = args[_s3 | (s1+s2)]; // always constructs s1+s2
    return s3;
}

std::string x = f((_s1="hello,", _s2=" world", _s3="hi world"));
```

Lazy Defaults

```
BOOST_PARAMETER_NAME(s1)
BOOST_PARAMETER_NAME(s2)
BOOST_PARAMETER_NAME(s3)

template <class ArgumentPack>
std::string f(ArgumentPack const& args)
{
    std::string const& s1 = args[_s1];
    std::string const& s2 = args[_s2];
    typename parameter::binding<
        ArgumentPack,tag::s3,std::string
    >::type
      s3 = args[_s3 || bind(std::plus<std::string>(), ref(s1), ref(s2)) ];
    return s3;
}`
```

creates a function object

```
std::string x = f((_s1="hello,", _s2=" world", _s3="hi world"));
```

Outline

- What’s the Point?
- Free Functions
- Member Functions
- Constructors & ArgumentPacks
- Class Templates
- Advanced Topics
  - Building ArgumentPacks
  - Extracting Parameter Types
  - Lazy Defaults
- **Best Practices**
- Python Binding

Best Practices: Keyword Naming

```
namespace people
{
  BOOST_PARAMETER_NAME(name)
  BOOST_PARAMETER_NAME(age)







  BOOST_PARAMETER_FUNCTION(
      (int), f, tag, (optional (name, *, "bob")(age, *, 42)))
  {
      std::cout << name << ":" << age;
      return age;
  }

  void test(int age)
  {
      f(_age = 3);
  }
}
```

Best Practices: Keyword Naming

```
namespace people
{
  namespace tag { struct name; struct age;  }
  namespace /* unnamed */ {
    boost::parameter::keyword<tag::name>& _name
    = boost::parameter::keyword<tag::name>::instance;
    boost::parameter::keyword<tag::age>& _age
    = boost::parameter::keyword<tag::age>::instance;
  }

  BOOST_PARAMETER_FUNCTION(
      (int), f, tag, (optional (name, *, "bob")(age, *, 42)))
  {
      std::cout << name << ":" << age;
      return age;
  }

  void test(int age)
  {
      f(_age = 3);
  }
}
```

Best Practices: Keyword Naming

```
namespace people
{
  namespace tag { struct name; struct age;  }
  namespace /* unnamed */ {
    boost::parameter::keyword<tag::name>& name
    = boost::parameter::keyword<tag::name>::instance;
    boost::parameter::keyword<tag::age>& age
    = boost::parameter::keyword<tag::age>::instance;
  }

  BOOST_PARAMETER_FUNCTION(
      (int), f, tag, (optional (name, *, "bob")(age, *, 42)))
  {
      std::cout << name << ":" << age;
      return age;
  }

  void test(int age)
  {
      f(age = 3);
  }
}
```

Best Practices: Namespaces

```
namespace people
{


  BOOST_PARAMETER_NAME(name)
  BOOST_PARAMETER_NAME(age)


  BOOST_PARAMETER_FUNCTION(
      (int), f, tag, (optional (name, *, "bob")(age, *, 42)))
  {
      std::cout << name << ":" << age;
  }
}

int x = people::f(people::_name = "jill", people::_age = 25);

using people::_name;  using people::_age;
int x = people::f(_name = "jill", _age = 25);

using namespace people;
int x = f(_name = "jill", _age = 25);
```

Best Practices: Namespaces

```
namespace people
{
  namespace keywords
  {
    BOOST_PARAMETER_NAME(name)
    BOOST_PARAMETER_NAME(age)
  }

  BOOST_PARAMETER_FUNCTION(
      (int), f, keywords::tag, (optional (name, *, "bob")(age, *, 42)))
  {
      std::cout << name << ":" << age;
  }
}

using namespace people::keywords;
int x = people::f(_name = "jill", _age = 25);
```

Outline

- What’s the Point?
- Free Functions
- Member Functions
- Constructors & ArgumentPacks
- Class Templates
- Advanced Topics
  - Building ArgumentPacks
  - Extracting Parameter Types
  - Lazy Defaults
- Best Practices
- **Python Binding**

Coming Full Circle

```
BOOST_PARAMETER_KEYWORD(tag, title)
BOOST_PARAMETER_KEYWORD(tag, width)
BOOST_PARAMETER_KEYWORD(tag, height)

class window
{
public:
  BOOST_PARAMETER_MEMBER_FUNCTION(
    (void), open, tag,
    (required (title, (std::string)))
    (optional
       (width, *(is_integral<_>), 400)
       (height, *(is_integral<_>), 400))
  )
  {
      … function implementation …
  }
};
```

```
struct open_fwd
{
  template <class A0, class A1, class A2>
  void operator()(
    boost::type<void>, window& self,
    A0 const& a0, A1 const& a1, A2 const& a2
  )
  {
    self.open(a0, a1, a2);
  }
};

 class_<window>("window")
   .def(
     "open",
     boost::python::function<
       open_fwd,
       mpl::vector<
         void,
         tag::title(std::string),
         tag::width*(unsigned),
         tag::height*(unsigned)
        >
     >()
    );
```

Bottom Lines

- Named-parameter interfaces
  - □ more self-documenting
  - □ easier to maintain
  - □ allow access to more useful defaults
- Deduced parameters work when types are sharply distinct
- ArgumentPacks allow sophisticated dispatching
- I want language support!
- And finally, don’t miss…

BoostCon 2009 – Aspen: May ?-?