Boost.Function

A Little Library's Co-Evolution with C++

Douglas Gregor, Assistant Director

Open Systems Lab @ Indiana University

What is Boost.Function?

- `boost::function` is a generalized function pointer
  - Same assignment/call/null-testing syntax as a function pointer
  - Can target any “compatible” function object

Basic Usage

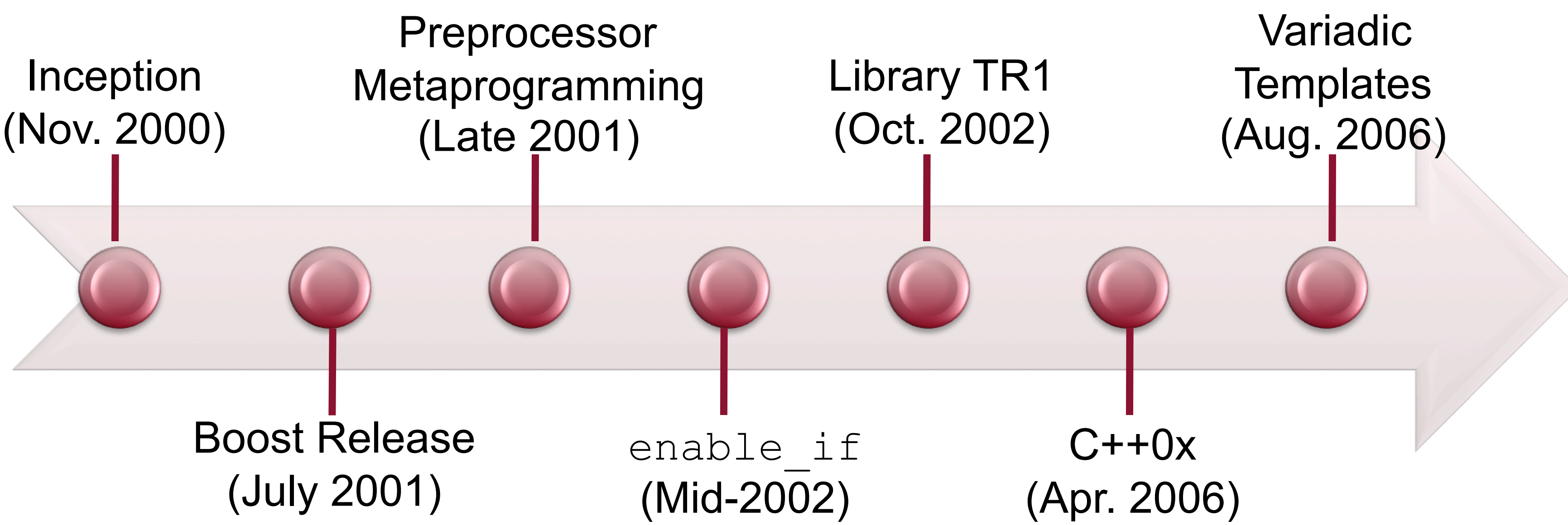
```
boost::function<int(int, int)> op;

op = std::plus<int>(); // assign value
cout << op(3, 5) << '\n'; // outputs 8

op = 0; // clear out the function
if (!op) { // check if the function is empty
  op = std::multiplies<long>(); // okay
}

if (op == std::multiplies<long>()) {
  // test what is inside the boost::function object
}
```

A Brief History of Function



Origins of Function (Nov. 2000)

- Boost.Threads needed a callback library
  - Bill Kempf defined the original requirements
- My “Blackrock” library tried to re-implement some Delphi features in C++
  - Properties and Events
- Boost.Function evolved from the Events part of Blackrock
  - and a competing library from Jesse Jones

Early interface (June 2001)

```
template<class Result, class T1 = unusable,
  class T2 = unusable, ..., class TN = unusable>
class function {
public:
  function();
  function(const function&);
  template<class F> function(const F&);
  function& operator=(const function&);
  template<class F> function& operator=(const F&);
  Result operator()(T1 a1, T2 a2, ..., TN aN);
  operator safe_bool() const;
  void clear();
};
```

Policies and Mixins

```
struct count_mixin { int depth_count; }
struct recursion_depth;

typedef function<int, int, int, mixin<count_mixin>,
                 policy<recursion_depth> > Func;
struct recursion_depth {

  void precall(Func* f) { ++f->depth_count; }
  void postcall(Func* f) { --f->depth_count; }
};
Func f; // tracks its recursion depth!
```

Implementation – Type Erasure

- `function` has always used type erasure:

```
template<class R, class T1, class T2>
struct invoker_base {
  virtual ~invoker_base();
  virtual R call(T1 a1, T2 a2) = 0;
}

template<class F, class R, class T1, class T2>
struct invoker : public invoker_base {
  virtual R call(T1 a1, T2 a2) { return f(a1, a2); }
  F f;
};
```

Implementation – Type Erasure

```
template<class R, class T1, class T2>
struct function {
  invoker_base<R, T1, T2> *invoker;

  template<class F> function(const F& f)
    : invoker(new invoker<F, R, T1, T2>(f)) { }

  R operator()(T1 a1, T2 a2) {
    return invoker->call(a1, a2);
  }
};
```

Optimization Issues

- Run-time cost of virtual functions is unavoidable
- Virtual functions cost space in a binary
  - Each invoker instance has a virtual constructor, virtual call, other functions.
- Type erasure without virtual functions:
  - Use function pointers instead
  - Significant reduction in executable size

Today’s Interface (July 2002)

- Function’s interface had always been somewhat clunky:

```
function<int, float> f;
```

- Peter Dimov suggested the use of function (pointer) syntax...

```
function<int(float)> f;
```

function and NULL

- Always wanted assignment, construction, and comparison from/with/to null pointer.
  - `if (f != 0) f = NULL;`
- Basic problem: general constructor eats NULL

```
template<class F> function(const F&);
```

  - Could add an “int” overload to catch 0, but...

`enable_if`/SFINAE (Mid 2002)

- Use of SFINAE for overloading discovered at this time

```
template<class F>
  function(const F&,
    typename enable_if<
                !is_integral<F>::value,
                void*>::type = 0);

function(secret_type*);
```

Preprocessor Metaprogramming

- Boost.Preprocessor library in late 2001
- Re-implemented Function using Boost.Preprocessor (Sep. 2002)
  - User's BOOST_FUNCTION_MAX_ARGS to an arbitrary number of arguments
  - Function's implementation became completely and totally unreadable

Library Technical Report 1

- The C++ committee initiated the first Library Extensions Technical Report (TR1) in October, 2002
- Function was one of the first libraries to go into TR1
  - function lost its `Allocator` parameter
- What was the other library?

Comparing Function Objects

- Most-requested feature [*]:
  - ```
    function<void(int)> f, g;
    if (f == g) { ... }
    ```
- Idea:
  - If `f` and `g` have the same type, compare them with `==`
  - Otherwise, `f != g`
- Unfortunately, it's unimplementable

[*] H. Sutter, *Generalizing Observer.C/C++ Users Journal, September, 2003.*
    ```

Type Erasure and `==`

- Problem: Arbitrary function objects cannot be compared with ==
- Analogy: `vector<T>`'s operator== works with those `T`'s that have an operator==.
  - Intuitively, `function`'s operator== should work like this
  - In reality, it can't: we need to know about == regardless of whether operator== is called

Implementing `operator==`

```
template<class R, class T1, class T2>
struct invoker_base {
  virtual bool compare(void *) const = 0;
  const std::type_info& type() = 0;
}

template<class F, class R, class T1, class T2>
struct invoker : public invoker_base {
  virtual bool compare(void * other)
    { return f == *(F*)other; }
  F f;
};
```

Implementing `operator==`

```
template<class R, class T1, class T2>
struct function {
  invoker_base<R, T1, T2> *invoker;

  template<class F> function(const F& f)
    : invoker(new invoker<F, R, T1, T2>(f)) { }

  const std::type_info& type();

  friend bool operator==(function& f1, function& f2)
  { if (f1.type() == f2.type())
      return f1.invoker->compare(&f2);
  }
};
```

Asymmetric Comparisons

- Asymmetric comparisons eliminate the type erasure:

```
template<typename Sig, typename F>
bool
operator==(const function<Sig>& f1, const F& f2) {
  return f1.type() == typeid(f2)
    && *f1.target<F>() == f2;
}
```

- These cover most use cases!

Allocators, Revisited

- Boost.Function has always had allocators

```
template<typename Signature, typename Allocator>
  class function;
```

- TR1 `function` has no allocator
- Emil Dotchevski introduced improved allocator support:

```
template <class F, class A>
  function( F f, A a );
```

Function in C++0x (Apr. 2006)

- C++ committee voted to move nearly all of Library TR1 into C++0x.
- Function was one of several Boost/TR1 libraries moved into C++0x at this time.
- C++0x features have forced some more evolution.

Variadic Templates (Aug. 2006)

- Function drove some design decisions:
  - Forced generalization of "a variable number of template arguments"
    - from "`tuple<T1, T2, ..., TN>`"
    - to "`R(T1, T2, ..., TN)`"
- Function benefits greatly from variadic templates
  - No more Preprocessor Metaprogramming!
  - Other C++0x features help, too.

C++0x Interface

```
template<class R, class... Args>
class function<R(Args...)> {
public:
  function();
  function(const function&);
  template<class F> function(const F&);
  function& operator=(nullptr_t);
  function& operator=(const function&);
  template<class F> function& operator=(const F&);
  Result operator()(Args... args);
  explicit operator bool() const;
};
```

Concepts Fix `operator==`! [*]

```
template<class F, class R, class... Args>
requires Callable<F, Args...>
struct invoker : public invoker_base {
  virtual bool compare(void *) { return false; }
};

template<class F, class R, class... Args>
requires Callable<F, Args...> && EqualityComparable<F>
struct invoker<F, R, Args...> : public invoker_base {
  virtual bool compare(void *other)
    { return f == *(F*)other; }
  F f;
};
```

[*] Mat Marcus noted the use of concepts to address the operator== problem.

Summary & Future of Function

- Function started in Boost
- It evolved as C++ evolved
- Boost.Function has “made it big”
  - TR1
  - C++0x