A Smart Function Pointer

Introduction

C++ does not have a standard general purpose "smart" function pointer despite C++ supporting various function types (standalone, member, and lambda/function objects). A smart function pointer should be capable of pointing to any of these types using a standard common type. Equality comparison should be supported. Function pointers have numerous uses (for example decoupling physical dependencies and implementing the observer pattern). So it is most unfortunate that a general purpose smart function pointer is not available that would be able to harmonize all functions types. Instead users resort to non-standard, complex templates, and class hierarchies to implement the needed features.

Why not std::function + std::bind?

It is true the combination of std::function + std::bind is able to call any of the function types. However std::function has constness problems (see N4159) and hopelessly lacks equality comparison because "it cannot be implemented "well"" (see boost faq). Equality comparison is a fundamental operation a smart function pointer needs to support. std::function also does not implement pointer semantics. Consider the following code:

```
void foo() { /*…*/ }

auto f1 = &foo;
auto f2 = f1;
for (int i = 0; i < 100; ++i) {
   if (isOdd(rand()))  f1();
   else f2();
}
```

With regular function pointers rand() does not have any effect – foo() is called 100 times. Now consider a similar example with std::function:

```
auto lam = /*…*/;
auto f1 = function<void()>(lam);
auto f2 = f1;
for (int i = 0; i < 100; ++i) {
   if (isOdd(rand())) f1();
   else f2();
}
```

Depending on the definition of the lambda function rand() may influence the program's behavior. std::function performs a deep copy which is not consistent with function pointer semantics. Function pointer semantics would indicate a shallow copy. This may not necessarily be a defect with std::function, but I am only pointing out that this is not pointer like semantics.

Additionally having to use std::function plus std::bind produces ugly syntax. There is a better way.

A very important use of std::function is to implement the observer pattern such as boost's signals2 library. Also see Generalizing Observer by Herb Sutter for a more thorough discussion of the observer pattern and problems with using std::function.

I am proposing a type that implements equality comparisons, has true function pointer like semantics, and provides a pleasant syntax. Substituting this type for std::function in the above code would produce results consistent with the function pointer example and can more cleanly implement the observer pattern.

Definitions

John Lakos has a talk on the meaning of values (see you tube: Value Semantics: It aint about the syntax). Basically value type should have 3 properties:

A. be able to substitutable with a copy
B. If A and B have the same values and the same salient operations are performed on a and b then both objects will again have the same value.
C. Two objects of a given value-semantic type have the same value if and only if there does not exist a distinguishing sequence among all of its salient operations.

The definition for a salient operation is basically any operation that has an effect on the type's value. The example Lakos gives is a std::vector's size is a salient attribute, but its capacity is not. So reserve() and shrink_to_fit() would not be salient operations but resize(), push_back(), etc. would be salient operations.

Smart Function Pointer Description

fun_ptr is a value type with its only salient attribute being its target. The target is the function (standalone, mutable member, const member, function object/lambda, or nullptr) and object instance when applicable. Unlike std::function (or boost::function) class fun_ptr is designed to have function pointer like semantics. fun_ptr supports equality operations. The target is invoked via operator(). Invoking a fun_ptr with a nullptr target is not permitted. operator= changes the target of the fun_ptr and is the only salient mutating operation that can be performed on a fun_ptr (note: operator() is not a salient operation).

Equality comparison. fun_ptrs with targets that point to the same object instance and function compare equal. Two fun_ptrs with different target types (i.e mutable member function and const member function) will compare unequal. Two fun_ptrs with different object instances will compare unequal. The equality comparison result is unspecified if the target type and object instances are the same but one or both of the member functions is virtual (because C++ spec does not specify the result of comparison of virtual function pointers).

Function object(or lambda) types are by convention (see Meyers Effective STL) passed by value (and often are temporary objects). When a fun_ptr is constructed with a function object it constructs a copy of the function object and then manages its lifetime. The function object will live so long as any fun_ptr references it. As a result two fun_ptrs constructed from the same function object will not compare equal (see example 1) because the targets are different instances of the function object type. However, copying a fun_ptr does not construct a new function object. The newly created fun_ptr will share the same target and extend the life of the function object should the original fun_ptr go out of scope.

For member function targets it is up to the user to ensure the target references a valid object. That is invoking a fun_ptr target on an object that no longer exists is not permitted. fun_ptr has shallow const and shallow copy semantics. Target function object lifetimes are managed by the fun_ptr class - not necessarily by a single fun_ptr

instance but possibly by any/all the fun_ptr instances - it is a shared target. The shared function object does not participate in the constness of fun_ptr.

The question might be asked: is fun_ptr thread safe? Recall that fun_ptr was designed to have function pointer like semantics so the answer is the same as the answer to the question: "is invoking a function pointer thread safe?". It depends on the implementation of the function its points to. For standalone functions the function would need to be reentrant. For a member function or function object it would need to be internally synchronized.

Smart Function Pointer Implementation

I have provided two separate implementations of fun_ptr. The first uses dynamic memory allocation and RTTI. The second uses the small object optimization for everything except function objects. The second implementation also uses a template trick is uses to remove RTTI. As a result the 2nd implementation it is on the order of 10x faster for most operations except lambdas.

Observer patter / Events

See Generalizing Observer by Herb Sutter for a more thorough discussion of the observer pattern and problems with using std::function.

With this smart function pointer and the use of operator== the observer pattern can be implemented without having to resort to cookies or handles to detach observers. Code like this is real and works! In my opinion this code has syntax on par with C# native implementations of delegates and events.

```
class High_score
{
public:
   //a public event object
   event<void(int)> newHighScoreEvent;

   High_score()
      : newHighScoreEvent()
      , newHighScoreEventInvoker(newHighScoreEvent.get_invoker())
      , high_score(0)
   {
   }

   void submit_score(int score)
   {
      if (score > high_score) {
         high_score = score;
         newHighScoreEventInvoker(high_score); //invoke the event
      }
   }

private:
   //a private invoker for the event (only this class may invoke the event)
   event<void(int)>::invoker_type newHighScoreEventInvoker;

   int high_score;
};
```

```
void print_new_high_score(int newHighScore)
{
   std::cout << "new high score = " << newHighScore << "\n";
}

int main()
{
   High_score hs;

   //can use a member function or lambda as well
   hs.newHighScoreEvent.attach(make_fun(&print_new_high_score));

   hs.submit_score(1);
   hs.submit_score(3);
   hs.submit_score(2);

   hs.newHighScoreEvent.detach(make_fun(&print_new_high_score));

   hs.submit_score(4);
}
```

References

Member Function Pointers and the Fastest Possible C++ Delegates By Don Clugston

The Impossibly Fast C++ Delegates By Sergey Ryazanov

std::function and Beyond (N4159) By Geoffrey Romer and Roman Perepelitsa

Value Semantics: It aint about the syntax! By John Lakos

Generalizing Observer by Herb Sutter

Smart Function Pointer Examples

Examples 1: function object equality

```
auto lam = []() { /*...*/ };
auto f1 = make_fun(lam);
auto f2 = f1; //copy constructor
assert(f2 == f1); //always true alfter a copy constructor
f1(); //invoke operator() const
assert(f2 == f1); //since operator() is const equality was not changed
f2 = make_fun(lam);
assert(f1 != f2); //a new instance of the lambda was created so f1 != f2
f1 = f2; //copy assignment
assert(f1 == f2); //always true alfter a copy assignment
```

Examples 2: member function object equality

```
auto f1 = make_fun(&Some_class::some_mem_fun, &some_class);
auto f2 = d1; //copy constructor
assert(f1 == f2); //always true alfter a copy constructor
f1(); //invoke operator() const
assert(f2 == f1); //since operator() is const equality was not changed
f2 = make_fun(&Some_class::some_mem_fun, &some_class);
assert(f1 == f2); //target is now the same so equal (note difference /w ex.1)
f1 = f2; //copy assignment
assert(f1 == f2); //always true alfter a copy assignment
```

fun_ptr Interface

Here is the interface for fun_ptr without implementation details.

```
#ifndef FUN_PTR_INTERFACE_H
#define FUN_PTR_INTERFACE_H

template <typename FuncSignature>
class fun_ptr;

template <typename Ret, typename... Args>
class fun_ptr<Ret(Args...)>
{
public:
   fun_ptr() noexcept;

   fun_ptr(std::nullptr_t) noexcept;

   // not declared noexcept because of a narrow contract and optionally dynamic
   // memory allocation is allowed.
   fun_ptr(Ret (*func)(Args...));

   // not declared noexcept because of a narrow contract and optionally dynamic
   // memory allocation is allowed.
   template <typename T>
   fun_ptr(Ret (T::*method)(Args...), T* object);

   // not declared noexcept because of a narrow contract and optionally dynamic
   // memory allocation is allowed.
   template <typename T>
   fun_ptr(Ret (T::*method)(Args...) const, const T* object);

   // not declared noexcept because optionally dynamic memory allocation is
   // allowed.
   template <typename T>
   fun_ptr(T f);

   fun_ptr(const fun_ptr& rhs) noexcept;

   fun_ptr(fun_ptr&& rhs) noexcept;

   ~fun_ptr() noexcept;

   fun_ptr& operator=(std::nullptr_t) noexcept;

   fun_ptr& operator=(const fun_ptr& rhs) noexcept;

   fun_ptr& operator=(fun_ptr&& rhs) noexcept;

   explicit operator bool() const noexcept;

   template <typename... FwArgs>
   Ret operator()(FwArgs... args) const;
};

template <typename Ret, typename... Args>
inline bool operator==(const fun_ptr<Ret(Args...)>& lhs, const fun_ptr<Ret(Args...)>&
rhs) noexcept;

template <typename Ret, typename... Args>
inline bool operator!=(const fun_ptr<Ret(Args...)>& lhs, const fun_ptr<Ret(Args...)>&
rhs) noexcept;
```

```
template <typename Ret, typename... Args>
inline bool operator==(const fun_ptr<Ret(Args...)>& lhs, std::nullptr_t) noexcept;

template <typename Ret, typename... Args>
inline bool operator!=(const fun_ptr<Ret(Args...)>& lhs, std::nullptr_t) noexcept;

template <typename Ret, typename... Args>
inline bool operator==(std::nullptr_t, const fun_ptr<Ret(Args...)>& rhs) noexcept;

template <typename Ret, typename... Args>
inline bool operator!=(std::nullptr_t, const fun_ptr<Ret(Args...)>& rhs) noexcept;

template <typename Ret, typename... Args>
inline fun_ptr<Ret(Args...)> make_fun(Ret (*fp)(Args...));

template <typename Ret, typename T, typename... Args>
inline fun_ptr<Ret(Args...)> make_fun(Ret (T::*fp)(Args...), T* obj);

template <typename Ret, typename T, typename... Args>
inline fun_ptr<Ret(Args...)> make_fun(Ret (T::*fp)(Args...) const, const T* obj);

template <typename T>
inline auto make_fun(T functor) -> decltype(make_fun(&T::operator(), (T*) nullptr));

#endif // FUN_PTR_INTERFACE_H
```

event Interface

```
template <typename FuncSignature>
class event;

template <typename Ret, typename... Args>
class event<Ret(Args...)>
{
public:
   event(const event&) = delete;
   event(event&&) = delete;
   event& operator=(const event&) = delete;
   event& operator=(event&&) = delete;

   typedef util3::fun_ptr<Ret(Args...)> delegate_type;

   typedef delegate_type invoker_type;

   event();

   //! @brief retrieve the invoker for this event.
   //!
   //! This function can only be called once.
   invoker_type get_invoker();

   void attach(const delegate_type& theDelegate);

   void detach(const delegate_type& theDelegate);
};
```