Accessing the target of a `tr1::function` object

Douglas Gregor

Document number: N1667=04-0107
Date: July 16, 2004
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor at cs.indiana.edu>

1 Introduction

Class template `tr1::function` stores function objects of arbitrary types. While these function objects are copied, invoked, and destroyed by the implementation as needed, their identities are completely lost to the user: there is no way to access the stored function object or determine its type.

Both of these abilities are desirable, especially when using `tr1::function` to build higher-level callback constructs such as delegates or signals/slots. In particular, these capabilities are required to implement delegates that can ignore duplicate targets and remove targets based on their function object values. For instance, one can imagine creating a `delegate` class template that operates like this (and is implemented with a container of `tr1::function` objects):

```
enum mouse_button { mb_left, mb_middle, mb_right };

delegate<void(int x, int y, mouse_button)> on_click;

on_click += display_context_menu();

// Say we want to record clicks for playback later...
on_click += record_macro(macro_name);

// We're done recording the macro: remove the function object
on_click -= record_macro(macro_name);
```

Here, the implementation of `-=` needs to compare function object wrappers for equality. Herb Sutter discusses these limitations in much more detail [1] and provides additional motivation for these features.

2 Target access

I propose to introduce two member functions to class template `tr1::function`, `type` and `target`. The `type` member function returns an `std::type_info` object referring to the type of the target function object (or `typeid(void)` if there is no target):

```
tr1::function<int(int, int)> f = std::plus<int>();
assert(f.type() == typeid(std::plus<int>));
tr1::function<int(int, int)> g;
assert(g.type() == typeid(void));
```

The `target` member function is templated on the type of the target and returns a pointer to the actual target function object (if the type matches) or a null pointer (if the type does not match), e.g.:

```
std::plus<int>* fp = f.target<std::plus<int> >(); // OK, fp points to stored object
std::minus<int>* nfp = f.target<std::minus<int> >(); // OK, NULL pointer
```

There are several alternatives to the member functions proposed, the most popular of which is some form of `operator==` for `tr1::function`. I am not proposing any variant of `operator==` because:

- We would still require the `type` and `target` member functions (or something like them) to have full access to the function object targets.
- `operator==` is unimplementable for `tr1::function` within the C++ language, because we do not have a reliable way to detect if a given type `T` is EQUALITY COMPARABLE without user assistance.
- A more limited form of `operator==`, which can compare a `tr1::function` object to any potential target, is implementable but less intuitive.[1] We need more experience before we can commit to such a strange form of operator.
- Adding `operator==` is not useful unless all of the standard binders also add `operator==`.

3 Proposed Text

Add to the end of the class definition in 3.4.3 [tr.func.wrap.func]:

```
// function target access
type_info type() const;
template<typename T> T* target();
template<typename T> const T* target() const;
```

Add a new subsection to 3.4.3 titled "function target access" [tr.func.wrap.func.target]:

```
type_info type() const;
```

Returns: If `*this` has a target of type `T`, `typeid(T)`; otherwise, `typeid(void)`.
Throws: will not throw.

```
template<typename T> T* target();
template<typename T> const T* target() const;
```

Requires: `T` must be a function object type callable with parameter types `T1`, `T2`, ..., `TN` and return type `R`.
Returns: If `type() == typeid(T)`, a pointer to the stored function target; otherwise, the NULL pointer.
Throws: will not throw.

References

[1] H. Sutter. Generalizing observer. *C/C++ Users Journal*, 21(9), September 2003. Available online at `http://www.cuj.com/documents/s=8840/cujexp0309sutter/`.

[1] Peter Dimov noted that this form of `operator==` is sufficient to implement delegates.