Document Number: to be reserved
Date: 2017-02-04
Audience: SG1
Author: Sergey Vidyuk

Possibility to erase std::packaged_task return type

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

1 Introduction

This document proposes to add partial specialization of the std::packaged_task class template which destroys information about the type returned by the function wrapped into the task. The main purpose of this specialization is to allow to store tasks with the same argument types, but different return types in the same collection.

Proof of concept implementation of the proposed feature is available on github [1]

2 Motivation

Implementing custom executor on top of the std::future API requires some type erasure class for function and function-like objects which:

- Hold write reference to a shared state of a future.
- Protect from accidental execution of the wrapped function more than once.
- Make shared state ready with some predictable exception if the object is destroyed without execution of the wrapped function.
- Store result of the wrapped function or any exception thrown in the shared state.
- Different objects wrapping functions with the same argument types, but different return type can be stored in a same collection.

Template class std::packaged_task satisfy all of those requirements except the last one. The worst way to overcome this limitation can look like this example:

[1] https://github.com/VestniK/portable_concurrency/tree/result-erased-task-proposal

```
// some thread-safe queue which is processed by some
// workers in multiple threads
using task_queue =
  mt_queue<std::function<void()>>;

template<typename F>
auto post_function(task_queue& queue, F&& func) {
  using R = std::result_of_t<F()>;
  auto task = std::make_shared<
    std::packaged_task<R()>
  >(func);
  auto res = task->get_future();
  queue.push([task = std::move(task)]() {
    (*task)();
  });
  return res;
}
```

This code introduces 2 extra allocations an 1 extra virtual call. Unfortunately I've seen the code like this more than once in a real life project.

The better solution is to create type erasure class satisfying requirements above. However, it's hard or impossible [2] to avoid 1 extra allocation and introduce 1 extra virtual call using this approach.

The best solution should be zero cost and perform no extra indirection or allocation.

3 Proposed solution

This document proposes to add tag-type std::ignore_t which is unusable for any other purposes and provide partial specialization of the std::packaged_task class template:

```
template<typename... A>
class packaged_task<ignore_t(A...)>;
```

[2] There are no requirements for std::packaged_task to have the same size and alignment for different instantiations. User code relying on such assumptions to avoid allocation can be broken by the compiler update.

It can be move constructed from std::packaged_task<R(A...)> with the same argument types and any return type. This partial specialization performs type-erasure of the task result type.

Proposed partial specialization has the same members with the same behavior as generic template with the following exceptions:

- No direct constructors from function or function-like objects provided.
- No get_future method provided.
- No reset method provided. [3]
- Provides constructor:

```
template<typename... A>
template<typename R>
packaged_task<ignore_t(A...)>::packaged_task(
  packaged_task<R(A...)>&& rhs
);
```

  with the following behavior:

  - Constructs a std::packaged_task with the shared state and task formerly owned by rhs, leaving rhs with no shared state and a moved-from task.
  - Throws exception of the type std::future_error with the code std::future_errc::broken_promise if rhs has shared state but the future pointing to it was not yet obtained via rhs.get_future().

- Provides assignment operator:

```
template<typename... A>
template<typename R>
packaged_task<ignore_t(A...)>::operator= (
  packaged_task<R(A...)>&& rhs
);
```

[3] There is no way to get a future to receive the result of the function wrapped in the std::packaged_task after reset is called on result-erased specialization so it's proposed to not provide this member for it.

with the following behavior:

- Releases the shared state, if any, destroys the previously-held task, and moves the shared state and the task owned by rhs into *this. rhs is left without a shared state and with a moved-from task.
- Throws exception of the type std::future_error with the code std::future_errc::broken_promise if rhs has shared state but the future pointing to it was not yet obtained via rhs.get_future().

Proposed result-erased partial specialization for the std::packaged_task allows to implement example from the motivation section in the following way:

```
// some thread-safe queue which is processed by some
// workers in multiple threads
using task_queue =
  mt_queue<std::packaged_task<std::ignore_t()>>;

template<typename F>
auto post_function(task_queue& queue, F&& func) {
  using R = std::result_of_t<F()>;
  auto task = std::packaged_task<R()>(func);
  auto res = task->get_future();
  queue.push(task);
  return res;
}
```

this code is simple, readable and deliver task to a worker without avoidable overhead.