4-coroutines

June 3, 2024

1 C++20 improved asynchronism with Coroutines

Coroutines are functions whose execution may be repeatedly suspended and resumed, with the possibility to exchange data each time.

C++ Coroutines are asymmetric and stackless: they suspend execution by returning to the caller, and the data that is required to resume execution is stored separately from the stack.

Currently, only a low level machinery is provided, so to allows library designers to implement higher-level tools such as generators, goroutines, tasks, and more.

Coroutines are well-suited for implementing event-driven applications or cooperative multitasking, and components such as iterators, infinite lists and pipes. BEWARE: they are usable only in environments where exceptions are forbidden or not available.

1.1 New keywords

A function is a **coroutine** if it uses some of the three new keywords : `co_await`, `co_yield` or `co_return`.

[1]:
```
%%file tmp.generator.h

#include <coroutine>

template <typename T>
struct generator
 {
  struct promise_type ;
  using coro_handle = std::coroutine_handle<promise_type> ;

  struct promise_type
   {
    std::suspend_always yield_value( T value ) { m_value = value ; return {} ; }
    std::suspend_always initial_suspend() { return {} ; }
    std::suspend_always final_suspend () { return {} ; }
    std::suspend_never return_void() { return {} ; }
    generator get_return_object () { return { coro_handle::from_promise(*this)␣
 ↪} ; }
    void unhandled_exception () { return ; }
    T m_value ;
```

```
   } ;

  coro_handle handle ;
  generator( coro_handle h ) : handle(h) {}
  ~generator() { if (handle) { handle.destroy() ; } }

  bool resume()
   {
    if (not handle.done()) { handle.resume() ; }
    return not handle.done() ;
   } ;
  T get()
   { return handle.promise().m_value ; }
 } ;
```

```
Writing tmp.generator.h
```

[2]:
```
%%file tmp.make-gen.h

#include "tmp.generator.h"

generator<int> make_gen( int start = 0, int step = 1 )
 {
  auto value = start ;
  while (true)
   {
    co_yield value ;
    value += step ;
   }
 }
```

```
Writing tmp.make-gen.h
```

[3]:
```
%%file tmp.coroutines.cpp

#include "tmp.make-gen.h"
#include <iostream>

int main()
 {
  auto gen = make_gen(100, -10) ;
  for ( int i = 0 ; i <= 5 ; ++i )
   { gen.resume() ; std::cout << gen.get() << " " ; }
  std::cout << std::endl;
 }
```

```
Writing tmp.coroutines.cpp
```

[4]:
```
!rm -f tmp.coroutines.exe && g++ -std=c++20 -fcoroutines -fno-exceptions tmp.
↪coroutines.cpp -o tmp.coroutines.exe
```

[5]:
```
!./tmp.coroutines.exe
```

```
100 90 80 70 60 50
```

1.2 Restrictions

- Incompatible with exceptions
- Coroutines cannot use: variadic arguments, plain return statements, placeholder return types (auto or concept).
- They cannot be coroutines: constexpr functions, constructors, destructors, main function.

1.3 Availability... almost here

- Visual C++ and Clang already support major portions, in the namespace `std::experimental`.
- GCC 10: coroutines isn't exposed with `-std=c++20`, but for now explicitly requires the `-fcoroutines -fno-exceptions` flags to be set.
- C++23 standard library should introduce turnkey elements such as `std::generator`.
- Meanwhile, a well known third-party library is https://github.com/lewissbaker/cppcoro.

1.4 Sources

- Cpp Reference
- C++ Coroutines Technical Specification
- Lewis Baker
- GCC Coroutines Wiki