# C++11 & Boost.Asio

NP TA 源灝

## Outline

1. Lambda Expressions
2. Auto Specifier
3. Shared Pointer
4. enable_shared_from_this
5. Move
6. Boost.Asio

## Lambda expressions (since C++11)

- An unnamed function object capable of capturing variables in scope.

```
/* without capture */
function<int(int)> square = [](int x) { return x * x; };
cout << square(5) << endl; /* output: 25 */


/* capture by reference */
int x = 0;
function<void(void)> increment = [&]() { ++x; };
cout << x << endl; /* output: 0 */
increment();
cout << x << endl; /* output: 1 */


/* capture by value */
function<void(void)> increment = [=]() { ++x; };
/* error: increment of read-only variable ‘x’ */
```

# Lambda expressions (since C++11)

Without Lambda Expression

```
bool by_first_name(Person a, Person b) {
  return a.first_name < b.first_name;
}

bool by_area(Shape a, Shape b) {
  return a.area < b.area;
}

/* sort employees ordered by first name */
vector<Person> employees;
sort(employees.begin(), employees.end(), by_first_name);

/* sort shapes ordered by area */
vector<Shape> shapes;
sort(shapes.begin(), shapes.end(), by_area);
```

## Lambda expressions (since C++11)

With Lambda Expression

```
/* sort employees ordered by first name */
vector<Person> employees;
sort(employees.begin(), employees.end(),[](Person a, Person b) {
  return a.first_name < b.first_name;
});

/* sort shapes ordered by area */
vector<Shape> shapes;
sort(shapes.begin(), shapes.end(), [](Shape a, Shape b) {
  return a.area < b.area;
});
```

## Auto Specifier (since C++11)

```
auto a = 1 + 2;     // int
auto b = a;         // int

/* function<int(int)> */
auto square = [](int x) { return x * x; };

vector<int> arr;
/* vector<int>::iterator */
auto begin_it = arr.begin();
```

# Shared Pointer (since C++11)

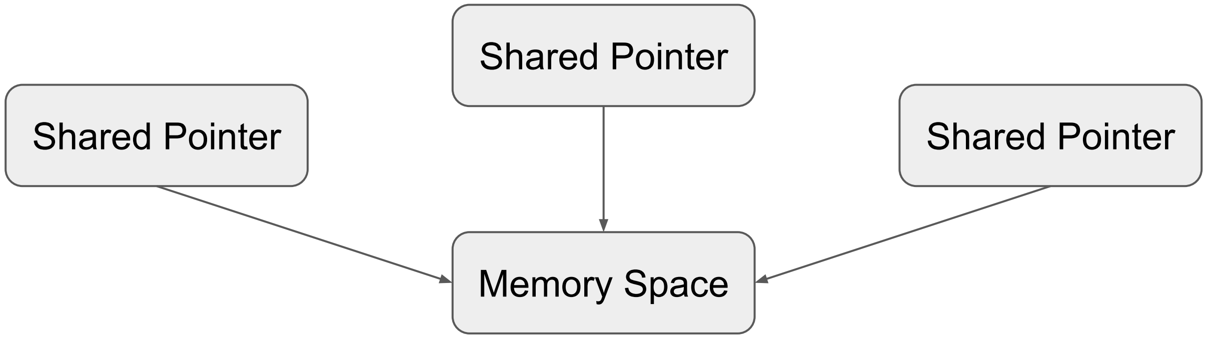
C++ smart pointers

```
1.  std::shared_ptr
2.  std::unique_ptr
3.  std::weak_ptr
```

- `std::shared_ptr`
- A **smart pointer** that retains shared ownership of an object through a pointer.
- You don't have to **free** or **delete** manually !

```
std::shared_ptr<myStruct> sp(new myStruct);
auto sp = std::make_shared<myStruct>();
```

- When will the object (myStruct) pointed by **sp be destroyed**?
  - The last remaining `shared_ptr` owning the object is destroyed (when **`use count == 0`**)



use count : 3

## enable_shared_from_this

- Allows an object `T` that is currently managed by a `shared_ptr` **safely generate additional `shared_ptr` instances.**

```
class MyClass : std::enable_shared_from_this<MyClass>
{
  std::shared_ptr<MyClass> get_ptr() {
    return shared_from_this(); // Correct
    return this;               // Wrong, DON'T do this!
  }
};
```

## Move (since C++11)

- `std::move` is used to indicate that an object t may be "moved from", i.e. allowing the efficient transfer of resources from t to another object.

```
string a = “Hello”;

/* extra cost of copying string a */
string b = a;

/* no string will be copyed, the content of
   string a will be moved into string c */
string c = move(a);

cout << ‘“‘ << a << ‘“‘ << endl;  // output: “”
cout << ‘“‘ << b << ‘“‘ << endl;  // output: “Hello”
cout << ‘“‘ << c << ‘“‘ << endl;  // output: “Hello”
```

## Boost.Asio C++11 Example (echo_server.cpp)

**[CAUTION]**

The codes on the slides are simplified. (e.g. namespaces are removed … )
They will not run without modification and adding the missing parts.

## Boost.Asio C++11 Example (echo_server.cpp)

```
io_service global_io_service;

int main(int argc, char* const argv[]) {
  short port = atoi(argv[1]);
  EchoServer server(port);
  global_io_service.run();
  return 0;
}
```

## Boost.Asio C++11 Example (echo_server.cpp)

```
global_io_service.run();
```

## Boost Asio io_service underlying mechanism

```
while (true) {
  select(max_fd + 1, &read_fdset, &write_fdset, NULL, NULL);
  for (int i = 0; i < all_fd.size(); ++i) {
    const int fd = all_fd[i];
    if (FD_ISSET(fd, &read_fdset)) {
      /* read data to buffer[] */
      auto done_callback = read_callback[fd];
      done_callback(buffer);
    }
    if (FD_ISSET(fd, &write_fdset)) {
      /* send data */
      auto done_callback = write_callback[fd];
      done_callback(length);
    }
  }
}
```

## Boost.Asio C++11 Example (echo_server.cpp)

```cpp
class EchoServer {
 private:
  ip::tcp::acceptor _acceptor;
  ip::tcp::socket _socket;
 public:
  EchoServer(short port)
      : _acceptor(global_io_service, port),
        _socket(global_io_service) {
    do_accept();
  }
 private:
  void do_accept() {
    _acceptor.async_accept(_socket, [this](error_code ec) {
      if (!ec)
        make_shared<EchoSession>(move(_socket))->start();
      do_accept();
    });
  }
};
```

global_io_service

## Boost.Asio C++11 Example (echo_server.cpp)

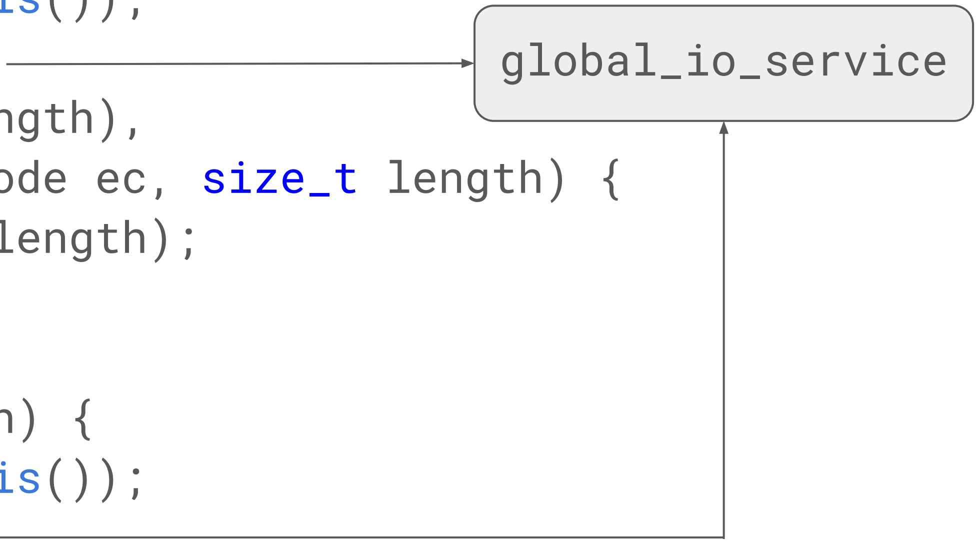
```
class EchoSession:public enable_shared_from_this<EchoSession>
{
 private:
  enum { max_length = 1024 };
  ip::tcp::socket _socket;
  array<char, max_length> _data;
 public:
  EchoSession(ip::tcp::socket socket):_socket(move(socket)){}
  void start() { do_read(); }

   .
   .
   .
   .
   .


}
```

# Boost.Asio C++11 Example (echo_server.cpp)

```
  ...
  void do_read() {
    auto self(shared_from_this());
    _socket.async_read_some(
        buffer(_data, max_length),
        [this, self](error_code ec, size_t length) {
          if (!ec) do_write(length);
        });
  }
  void do_write(size_t length) {
    auto self(shared_from_this());
    _socket.async_send(
        buffer(_data, length),
        [this, self](error_code ec, size_t length) {
          if (!ec) do_read();
        });
  }
}
```



# Project 3

```
class ShellSession : enable_shared_from_this<ShellSession> {
 private:
  /* ... some data members */
 public:
  start() { do_resolve(); }
 private:
  do_resolve() { async_resolve(..., do_connect); }
  do_connect() { async_connect(..., do_read); }
  do_read() {
    async_read(
      ...,
      []() {
        if (buffer contains "% ")
          do_send_cmd();
        do_read();
      });
  }
  do_send_cmd() { ... }
};
```