NPPJ3 supplementary

NP TA 子頤

Outline

1. auto
2. smart pointers
3. lambda expression
4. `echo_server.cpp` explained

Recap: C++'s history[1]

Year	C++ Standard	Informal name
1998	ISO/IEC 14882:1998	C++98
2003	ISO/IEC 14882:2003	C++03
2011	ISO/IEC 14882:2011	C++11, C++0x
2014	ISO/IEC 14882:2014	C++14, C++1y
2017	ISO/IEC 14882:2017	C++17, C++1z
2020	to be determined	C++20, C++2a

Modern C++

- `template type deduction`
- `auto, decltype`
- `smart pointers`
- `lambda`
- `uniform initializer`
- `constexpr`

and much more, for just C++11!
Not to mention C++14/17…

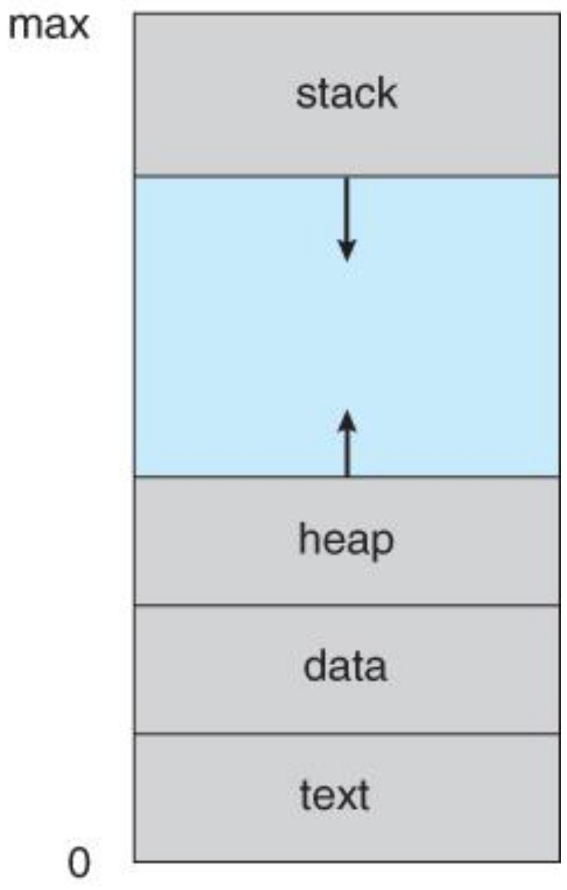
[1]:https://en.wikipedia.org/wiki/C%2B%2B

New feature 1. auto

Let the compiler help you deduce types:

```
auto x1 = 0;                                // x1's type is int

auto x2 = 0.3f;                             // x2's type is float

auto vec1 = vector<int>{1, 2, 3, 4, 5};     // vec1's type is vector<int>

auto sp = make_shared<int>(1234);           // sp type is shared_ptr<int>

for (auto it = vec1.begin(); it != vec1.end(); ++it) {
   cout << *it << ' ';
}
```

New feature 2. Smart Pointers



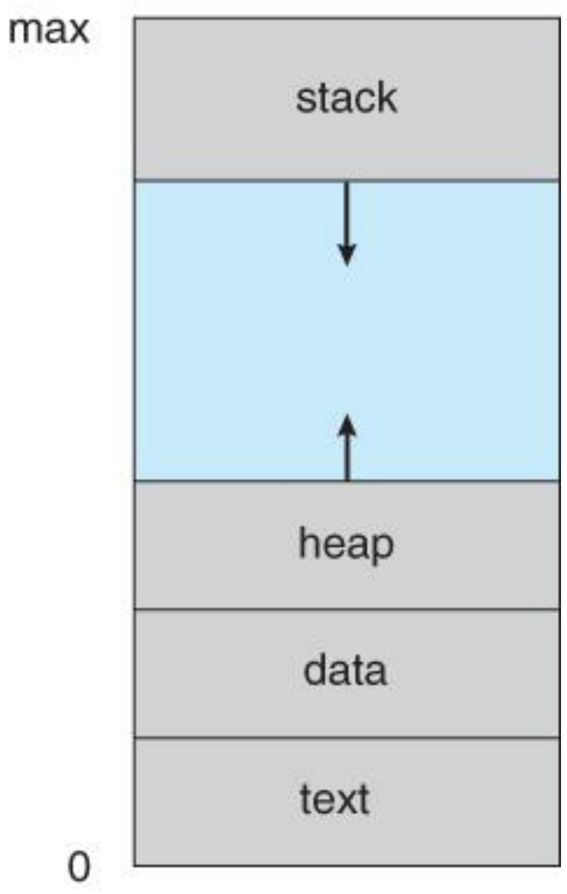
local variables, function parameters, …

```
int a;
char str[] = "String";
```

Dynamic Memory Allocation

```
int* pInt = new int;
char* str = new char[100];

delete[] str;
delete pInt;
```

New feature 2. Smart Pointers



local variables, function parameters, …

```
int a;
char str[] = "String";
```

Dynamic Memory Allocation

```
int* pInt = new int;
char* str = new char[100];
```

```
delete[] str;
delete pInt;
```

Memory leak if you forgot to delete!

```
#include <memory>
```

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

Smart Pointers :: unique_ptr

```
{
    unique_ptr<int> ptr(new int);   // allocated a pointer to int
    cin >> *ptr;                    // Use * to access underlying resource
    cout << *ptr;

} // Automatically delete the resource following the destruction of
  // unique_ptr.
```

On construction:

On destruction:

No resources are leaked!

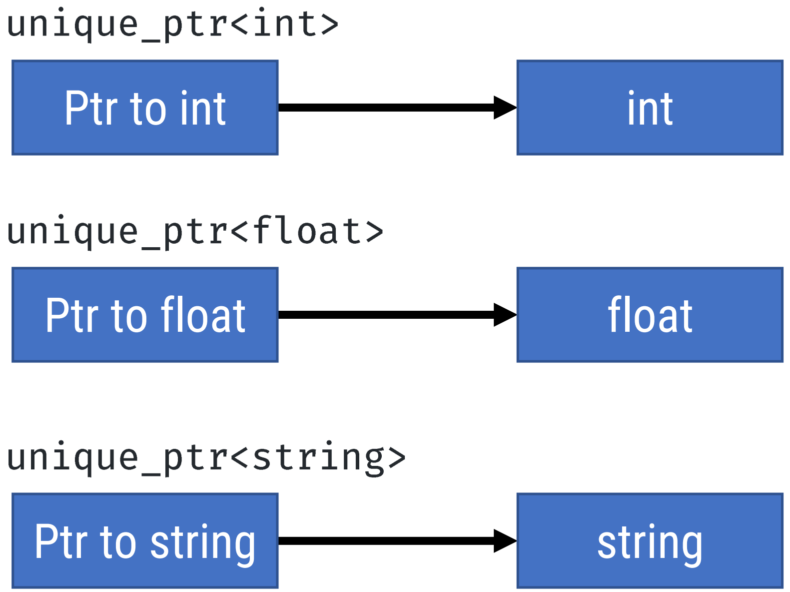
```cpp
#include <memory>
```

1. **unique_ptr**
2. shared_ptr
3. weak_ptr

Note:

- unique_ptr is a **class template**, which means you can construct it with **any** type:

```cpp
unique_ptr<int> ptr(new int);

unique_ptr<float> ptr(new float);

unique_ptr<string> ptr(new string);

unique_ptr<vector<int>>  ptr(new vector<int>);
```



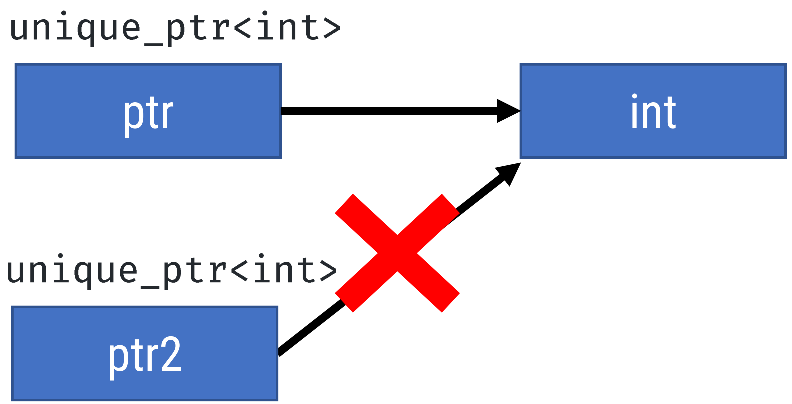
```
#include <memory>
```

1. **unique_ptr**
2. shared_ptr
3. weak_ptr

Note:

➢ unique_ptr is **move-only**, copy-assignments are not allowed.

```
unique_ptr<int> ptr(new int);
unique_ptr<int> ptr2;
ptr2 = ptr;            // Error!! Sharing ptr's resource with ptr2 is not
                       // allowed.
```



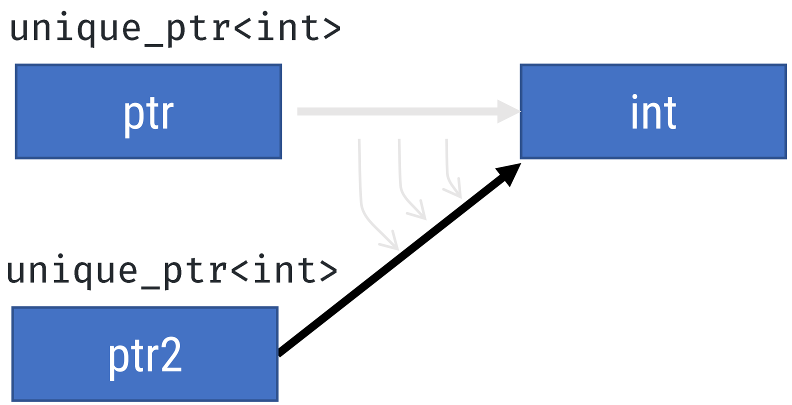
```
#include <memory>
```

1. **unique_ptr**
2. shared_ptr
3. weak_ptr

Note:

- unique_ptr is **move-only**, copy-assignments are not allowed.

```
unique_ptr<int> ptr(new int);
unique_ptr<int> ptr2;
ptr2 = std::move(ptr);  // the ownership of the allocated int is moved from
                        // ptr to ptr2. ptr no longer owns the resource.
```



```
#include <memory>
```

1. unique_ptr
2. shared_ptr
3. weak_ptr

Note:

➢ unique_ptr is **move-only**, copy-assignments are not allowed.

```
unique_ptr<int> ptr(new int);
unique_ptr<int> ptr2;
ptr2 = std::move(ptr); // the ownership of the allocated int is moved from
                       // ptr to ptr2. ptr no longer owns the resource.
```

Can we shared the resource by multiple pointers?

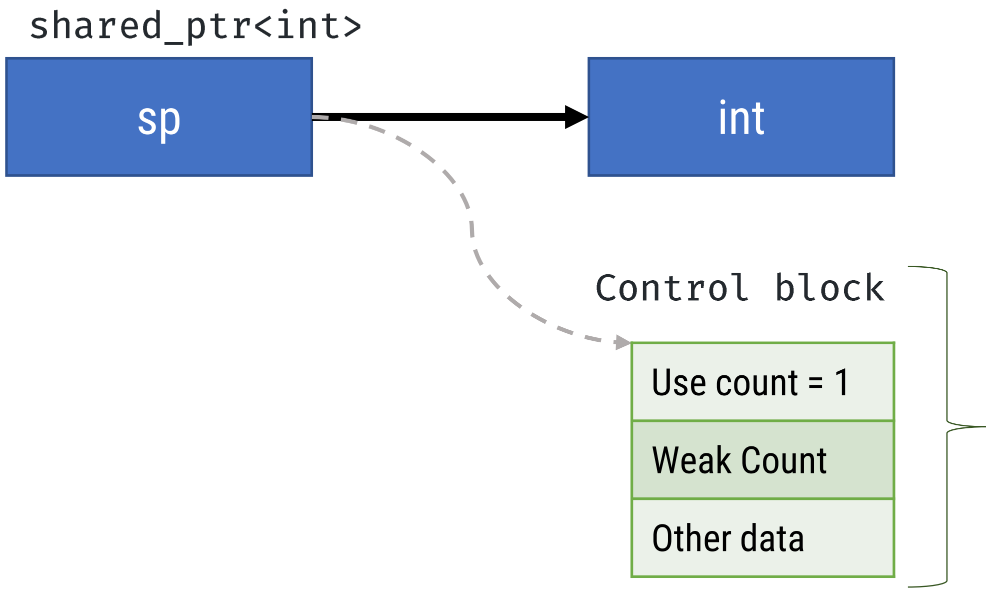
(Yes!)

```cpp
#include <memory>
```

1. unique_ptr
2. **shared_ptr**
3. weak_ptr

Smart Pointers :: shared_ptr

```cpp
{
    shared_ptr<int> sp(new int);      // create a shared pointer to int
    cout << sp.use_count();           // print 1
    cin >> *sp;                       // Use * to access underlying resource
    cout << *sp;
}
```



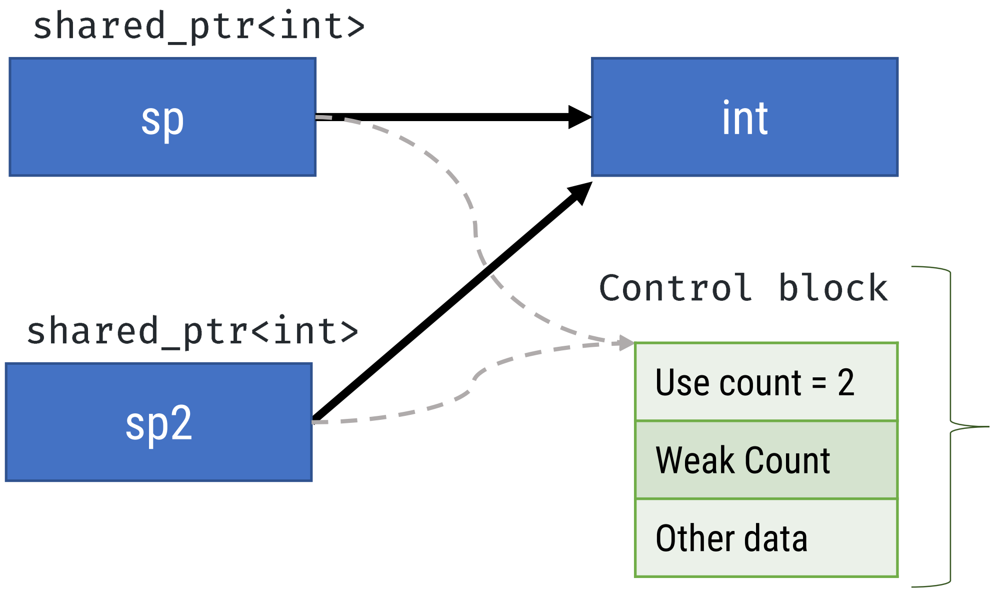
Additional info used by shared_ptr.

```
#include <memory>
```

Smart Pointers :: shared_ptr

1. unique_ptr
2. **shared_ptr**
3. weak_ptr

```
{
    shared_ptr<int> sp(new int);     // create a shared pointer to int
    shared_ptr<int> sp2;             // create an empty shared ptr
    sp2 = sp;                        // sp “shared” it’s resource with sp2
    cout << sp2.use_count();         // print 2
} // The resource will be deleted when use_count = 0
```



Additional info used by shared_ptr.

```cpp
#include <memory>
```

Smart Pointers :: shared_ptr

1. unique_ptr
2. **shared_ptr**
3. weak_ptr

```cpp
{
    shared_ptr<int> sp(new int);        // Allocate resource, use_cnt = 1
    {
        shared_ptr<int> sp2(sp);        // sp shared it’ s resource with sp2,
                                        // use_cnt += 1

        cout << sp.use_count();         // print 2

    }                                   // sp2 destroyed, use_cnt -= 1

    cout << sp.use_count();             // print 1

}                                       // sp destroyed, use_cnt -= 1,
                                        // since use_cnt = 0, the allocated
                                        // resource is deleted.
```

```cpp
#include <memory>
```

1. unique_ptr
2. **shared_ptr**
3. weak_ptr

std::make_shared<T>

➢ A factory function to generate shared_ptr:

```cpp
auto sp = std::make_shared<int>(7);                // make a shared_ptr to int
cout << *sp;                                       // print 7

auto sp2 = std::make_shared<string>("string");     // make a shared_ptr to string
cout << *sp2;                                      // print “string”
```

shared_ptr pitfalls

```
class Widget
{
    int i_;
    float f_;
};
```

Use count are not correct! Why?

```
{
    Widget* pW = new Widget;
    shared_ptr<Widget> spw(pW);
    shared_ptr<Widget> spw2(pW);
    cout << spw.use_count();          // 1 (!)
    cout << spw2.use_count();         // 1 (!)
}
```

The following **rules** for control block creation are used[1]:

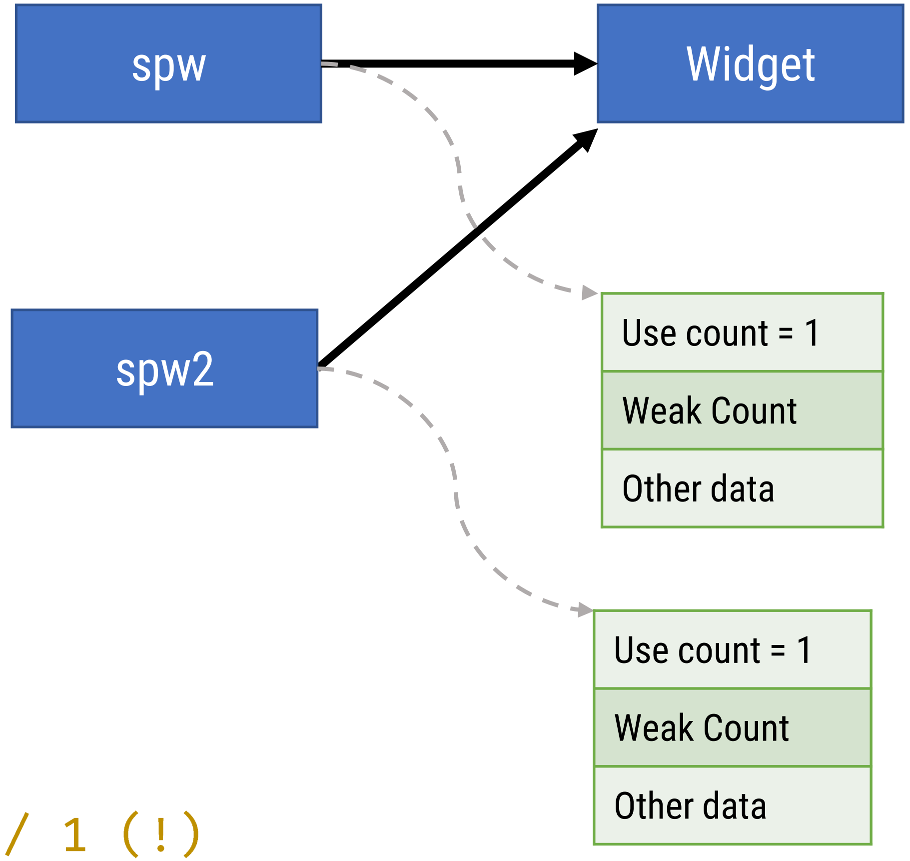
1. `std::make_shared` always creates a control block.
2. A control block is created when a `std::shared_ptr` is constructed from an unique-ownership pointer (i.e., `std::unique_ptr`).
3. When a `std::shared_ptr` constructor is called with a raw pointer, it creates a control block.

[1]: 《Effective Modern C++》 item 19

shared_ptr pitfalls

```
class Widget
{
    int i_;
    float f_;
};


{
    Widget* pW = new Widget;
    shared_ptr<Widget> spw(pW);
    shared_ptr<Widget> spw2(pW);
    cout << spw.use_count();        // 1 (!)
    cout << spw2.use_count();       // 1 (!)
}
```

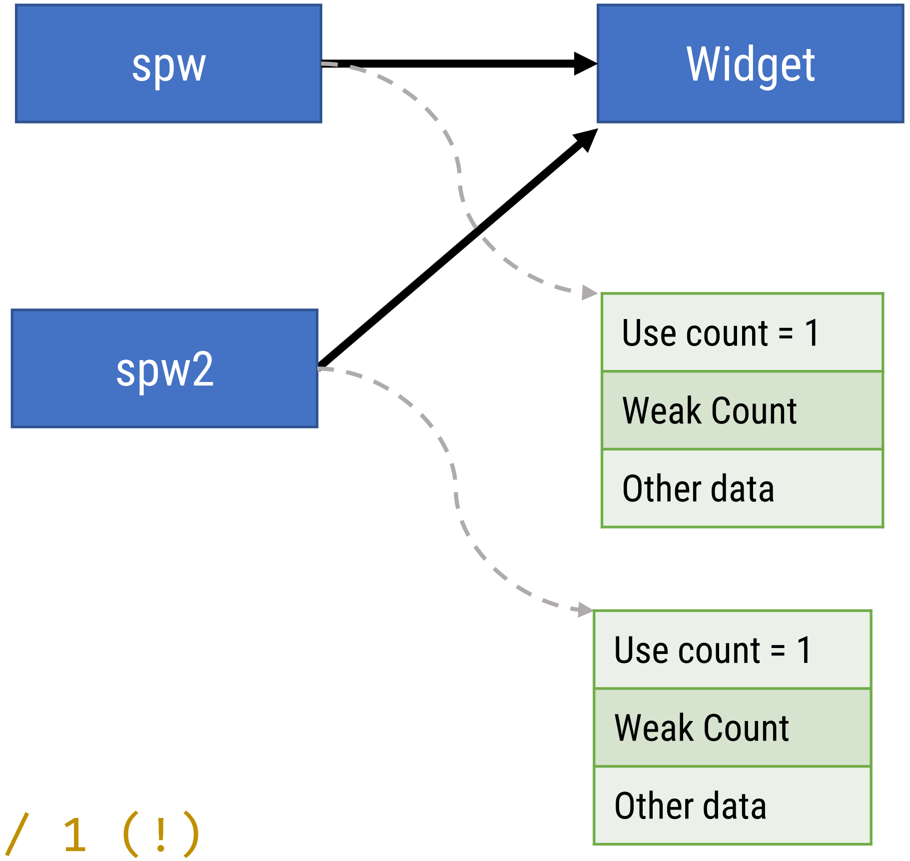
- The 3rd rule is met, so a control block is created



shared_ptr pitfalls

```
class Widget
{
    int i_;
    float f_;
};


{
    Widget* pW = new Widget;
    shared_ptr<Widget> spw(pW);
    shared_ptr<Widget> spw2(pW);
    cout << spw.use_count();        // 1 (!)
    cout << spw2.use_count();       // 1 (!)
}
```

- The 3rd rule is met, so a control block is created



Workaround?

shared_from_this()

Enable `shared_from_this()` for class Widget.

```
class Widget : public enable_shared_from_this<Widget>
{
    int i_;
    float f_;
};
```

`shared_from_this()` returns a shared_ptr that points to the same object as `spw` does.

```
{
    Widget* pW = new Widget;
    shared_ptr<Widget> spw(pW);
    shared_ptr<Widget> spw2(pW->shared_from_this());
    cout << spw.use_count();
    cout << spw2.use_count();
}
```

shared_from_this()

Enable `shared_from_this()` for class Widget.

```
class Widget : public enable_shared_from_this<Widget>
{
    int i_;
    float f_;
};
```

`shared_from_this()` returns a shared_ptr that points to the same object as `spw` does.

```
{
    Widget* pW = new Widget;
    shared_ptr<Widget> spw(pW);
    shared_ptr<Widget> spw2(pW->shared_from_this());
    cout << spw.use_count();
    cout << spw2.use_count();
}
```

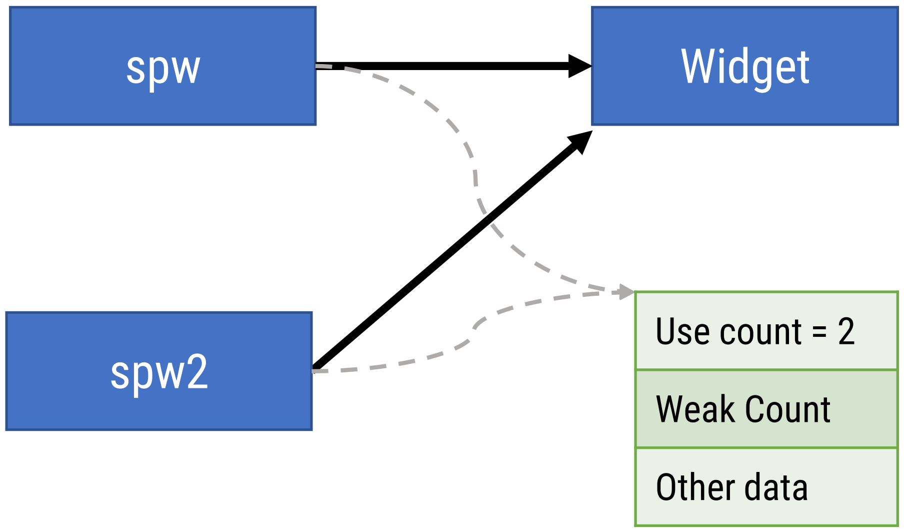
`spw2` is now constructed by a shared_ptr, thus will not create another control block.

`#include <memory>`

1. unique_ptr
2. **shared_ptr**
3. weak_ptr

shared_from_this()

```
class Widget : public enable_shared_from_this<Widget>
{
    int i_;
    float f_;
};
```




```
{
    Widget* pW = new Widget;
    shared_ptr<Widget> spw(pW);
    shared_ptr<Widget> spw2(pW->shared_from_this());
    cout << spw.use_count();
    cout << spw2.use_count();
}
```

shared_from_this()

```
class Widget : public enable_shared_from_this<Widget>
{
   int i_;
   float f_;

   shared_ptr<Widget> get_this() {
     return shared_from_this();
   }
};
```

If `shared_from_this()` is invoked **inside** the class, it returns a shared_ptr that points to the same object as the `this` pointer does.

New feature 3. Lambda

A lambda is composed of **three** parts: **[ ] ( ) { }**

- *capture clause*
- *parameter list*
- *lambda body*

e.g.,

```
auto add = [](int lhs, int rhs) { return lhs + rhs; };
cout << add(3, 5);  // print 8
cout << add(-3, 7); // print 4
```

e.g.,

```
vector<int> vec{100, 200, 50, -100, 4, -500};
cout << find_if(vec.begin(), vec.end(), [](int val)
               { return 0 < val && val < 10; }) - vec.begin(); // print 4
```

Don’t worry!

➢ Though lambda’s syntax is quite ugly, it turns out to be a time-saver when you adapt to use it :

```
find_if(vec.begin(), vec.end(), [](int val) { return 0 < val && val < 10; })
```

vector<int> vec =

0	1	2	3	4	5
100	200	50	-100	4	-500

Lambda returns false

Don’t worry!

- Though lambda’s syntax is quite ugly, it turns out to be a time-saver when you adapt to use it :

```
find_if(vec.begin(), vec.end(), [](int val) { return 0 < val && val < 10; })
```

vector<int> vec =

0	1	2	3	4	5
100	200	50	-100	4	-500

Lambda returns false

Don’t worry!

- Though lambda’s syntax is quite ugly, it turns out to be a time-saver when you adapt to use it :

```
find_if(vec.begin(), vec.end(), [](int val) { return 0 < val && val < 10; })
```

vector<int> vec =

0	1	2	3	4	5
100	200	50	-100	4	-500

Lambda returns false

Don’t worry!

- Though lambda’s syntax is quite ugly, it turns out to be a time-saver when you adapt to use it :

```
find_if(vec.begin(), vec.end(), [](int val) { return 0 < val && val < 10; })
```

vector<int> vec =

0	1	2	3	4	5
100	200	50	-100	4	-500

Lambda returns false

Don’t worry!

- Though lambda’s syntax is quite ugly, it turns out to be a time-saver when you adapt to use it :

```
find_if(vec.begin(), vec.end(), [](int val) { return 0 < val && val < 10; })
```

vector<int> vec =

0	1	2	3	4	5
100	200	50	-100	4	-500

Lambda returns true

Don’t worry!

➢ Though lambda’s syntax is quite ugly, it turns out to be a time-saver when you adapt to use it :

```
find_if(vec.begin(), vec.end(), [](int val) { return 0 < val && val < 10; })
```

vector<int> vec =

0	1	2	3	4	5
100	200	50	-100	4	-500

Lambda returns true

The iterator pointed to the 5th element is returned

Alternatives? (How to do the same job without lambda)

- Pass a **function**

```
bool inside(int val){
  return 0 < val && val < 10;
}


find_if(vec.begin(), vec.end(), inside);
```

- Downside:

  Many functions are therefore created and results in code bloat.

Alternatives? (How to do the same job without lambda)

- Pass a **function object**

```
class Functor {
  void operator()(int val) {
   return 0 < val && val < 10;
  }
};
find_if(vec.begin(), vec.end(), Functor());
```

- Downside:

  Many classes are therefore created and results in code bloat.

To “**see**” variables outside lambda

- Capture it by value

```
int v = 7;
auto add = [v] (int rhs) { return v + rhs; };
cout << add(3);      // print 10
cout << v;           // print 7
```

- Capture by reference

```
int v = 7;
auto add = [&v] (int rhs) { return v + rhs; };
cout << add(3);      // print 10
cout << v;           // print 7
```

...and to “**modify**” their value:

- If captured by value, make it **mutable:**

```
int v = 7;
auto add = [v] (int rhs) mutable { v = 4; return v + rhs; };
                          // However, this doesn’t affect external v’s value
cout << add(3);           // print 7
cout << v;                // print 7
```

- If captured by reference, just modify it (also affects the original one):

```
int v = 7;
auto add = [&v] (int rhs) { v = 4; return v + rhs; };
cout << add(3);           // print 7
cout << v;                // print 4
```

In echo_server.cpp, lambdas are been used as a handler.

```cpp
  void do_accept() {
    _acceptor.async_accept(_socket, [this](boost::system::error_code ec) {
      if (!ec) make_shared<EchoSession>(move(_socket))->start();

      do_accept();
    });
  }
};
```

- ➢ We know that `async_accept()` returns immediately, and the handler (lambda) will be called when an accept operation complete. At that moment, if there is no error, a `shared_ptr` to `EchoSession` will be created and `start()` will be called.

```cpp
22    void start() { do_read(); }
23
24   private:
25    void do_read() {
26      auto self(shared_from_this());
27      _socket.async_read_some(
28          buffer(_data, max_length),
29          [this, self](boost::system::error_code ec, size_t length) {
30            if (!ec) do_write(length);
31          });
32    }
```

- In `do_read()`, we create a `shared_ptr` to `*this`, named `self`, and is captured by the handler. If you don't capture `self`, the allocated `EchoSession` (line 59) will be deleted.

  (Because as the execution flow continuous, it will return from do_read(), start(), and before calling do_accept(), there is no more shared_ptr pointing to that EchoSession, thus results in the deletion).