Lecture 11.26

Smart Pointer (2)

SE271 Object-Oriented Programming (2020)

Yeseong Kim

Original slides: Univ. of Washington, CSE333, Spring 2018

Short Notice

- Team Project – Released the instruction
  - Will have a presentation with a recorded video (4 minutes for each team)
  - Will write a report (3~5 pages)

  - 1. Final demo video: 12월 6일 (토, 자정) 비중: 35%
  - 2. Report / Source code: 12월 19일 (일, 자정) 비중: 60%

Today’s Topic

- Smart Pointer
  - unique_ptr
  - shared_ptr
  - weak_ptr

`std::unique_ptr`

- A `unique_ptr` *takes ownership* of a pointer
  - Part of C++'s standard library (C++11)
  - Its destructor invokes `delete` on the owned pointer
    - Invoked when `unique_ptr` object is `delete`'d or falls out of scope

Transferring Ownership

- Use **reset**() and **release**() to transfer ownership
  - **release** returns the pointer, sets wrapper's pointer to NULL
  - **reset** delete's the current pointer and stores a new one

```
int main(int argc, char **argv) {
  unique_ptr<int> x(new int(5));
  cout << "x: " << x.get() << endl;

  unique_ptr<int> y(x.release());  // x abdicates ownership to y
  cout << "x: " << x.get() << endl;
  cout << "y: " << y.get() << endl;

  unique_ptr<int> z(new int(10));

  // y transfers ownership of its pointer to z.
  // z's old pointer was delete'd in the process.
  z.reset(y.release());

  return EXIT_SUCCESS;
}
```

`unique_ptr` and STL

- `unique_ptr`s *can* be stored in STL containers
  - Wait, what? STL containers like to make lots of copies of stored objects and `unique_ptr`s cannot be copied…

- Move semantics to the rescue!
  - When supported, STL containers will *move* rather than *copy*
    - `unique_ptr`s support move semantics

Aside: Copy Semantics

- Assigning values typically means making a copy
  - Sometimes this is what you want
    - *e.g.* assigning a string to another makes a copy of its value
  - Sometimes this is wasteful
    - *e.g.* assigning a returned string goes through a temporary copy

```
std::string ReturnFoo(void) {
  std::string x("foo");
  return x;   // this return might copy
}

int main(int argc, char **argv) {
  std::string a("hello");
  std::string b(a);  // copy a into b

  b = ReturnFoo();   // copy return value into b

  return EXIT_SUCCESS;
}
```

Transferring Ownership via Move

- unique_ptr supports move semantics
  - Can "move" ownership from one unique_ptr to another
    - Behavior is equivalent to the "release-and-reset" combination

```
int main(int argc, char **argv) {
  unique_ptr<int> x(new int(5));
  cout << "x: " << x.get() << endl;

  unique_ptr<int> y = std::move(x); // x abdicates ownership to y
  cout << "x: " << x.get() << endl;
  cout << "y: " << y.get() << endl;

  unique_ptr<int> z(new int(10));

  // y transfers ownership of its pointer to z.
  // z's old pointer was delete'd in the process.
  z = std::move(y);

  return EXIT_SUCCESS;
}
```

unique_ptr and STL Example

uniquevec.cc

```
int main(int argc, char **argv) {
  std::vector<std::unique_ptr<int> > vec;

  vec.push_back(std::unique_ptr<int>(new int(9)));
  vec.push_back(std::unique_ptr<int>(new int(5)));
  vec.push_back(std::unique_ptr<int>(new int(7)));

  //
  int z = *vec[1];
  std::cout << "z is: " << z << std::endl;

  //
  std::unique_ptr<int> copied = vec[1];

  //
  std::unique_ptr<int> moved = std::move(vec[1]);
  std::cout << "*moved: " << *moved << std::endl;
  std::cout << "vec[1].get(): " << vec[1].get() << std::endl;

  return EXIT_SUCCESS;
}
```

`unique_ptr` and "<"

- A `unique_ptr` implements some comparison operators, including `operator<`
  - However, it doesn't invoke `operator<` on the pointed-to objects
  - So to use **`sort`**`()` on `vector`s, you want to provide it with a comparison function

unique_ptr and STL Sorting

uniquevecsort.cc

```
using namespace std;
bool sortfunction(const unique_ptr<int> &x,
                  const unique_ptr<int> &y) { return *x < *y; }
void printfunction(unique_ptr<int> &x) { cout << *x << endl; }

int main(int argc, char **argv) {
  vector<unique_ptr<int> > vec;
  vec.push_back(unique_ptr<int>(new int(9)));
  vec.push_back(unique_ptr<int>(new int(5)));
  vec.push_back(unique_ptr<int>(new int(7)));

  // buggy: sorts based on the values of the ptrs
  sort(vec.begin(), vec.end());
  cout << "Sorted:" << endl;
  for_each(vec.begin(), vec.end(), &printfunction);

  // better: sorts based on the pointed-to values
  sort(vec.begin(), vec.end(), &sortfunction);
  cout << "Sorted:" << endl;
  for_each(vec.begin(), vec.end(), &printfunction);

  return EXIT_SUCCESS;
}
```

`unique_ptr` and Arrays

- `unique_ptr` can store arrays as well
  - Will call `delete[]` on destruction

unique5.cc

```
#include <memory>    // for std::unique_ptr
#include <cstdlib>   // for EXIT_SUCCESS

using namespace std;

int main(int argc, char **argv) {
  unique_ptr<int[]> x(new int[5]);

  x[0] = 1;
  x[2] = 2;

  return EXIT_SUCCESS;
}
```

`std::shared_ptr`

- `shared_ptr` is similar to `unique_ptr` but we allow shared objects to have multiple owners
  - The copy/assign operators are not disabled and *increment* a reference count
    - After a copy/assign, the two `shared_ptr` objects point to the same pointed-to object and the (shared) reference count is 2
  - When a `shared_ptr` is destroyed, the reference count is *decremented*
    - When the reference count hits 0, we `delete` the pointed-to object!

`shared_ptr` Example

sharedexample.cc

```
#include <cstdlib>    // for EXIT_SUCCESS
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::shared_ptr

int main(int argc, char **argv) {
  std::shared_ptr<int> x(new int(10));  // ref count:

  // temporary inner scope (!)
  {
    std::shared_ptr<int> y = x;         // ref count:
    std::cout << *y << std::endl;
  }

  std::cout << *x << std::endl;         // ref count:

  return EXIT_SUCCESS;
}                                       // ref count:
```

shared_ptrs and STL Containers

- Even simpler than unique_ptrs
  - Safe to store shared_ptrs in containers, since copy/assign maintain a shared reference count

sharedvec.cc

```
vector<std::shared_ptr<int> > vec;

vec.push_back(std::shared_ptr<int>(new int(9)));
vec.push_back(std::shared_ptr<int>(new int(5)));
vec.push_back(std::shared_ptr<int>(new int(7)));

int &z = *vec[1];
std::cout << "z is: " << z << std::endl;

std::shared_ptr<int> copied = vec[1];  // works!
std::cout << "*copied: " << *copied << std::endl;

std::shared_ptr<int> moved = std::move(vec[1]);  // works!
std::cout << "*moved: " << *moved << std::endl;
std::cout << "vec[1].get(): " << vec[1].get() << std::endl;
```

Cycle of `shared_ptrs`

strongcycle.cc

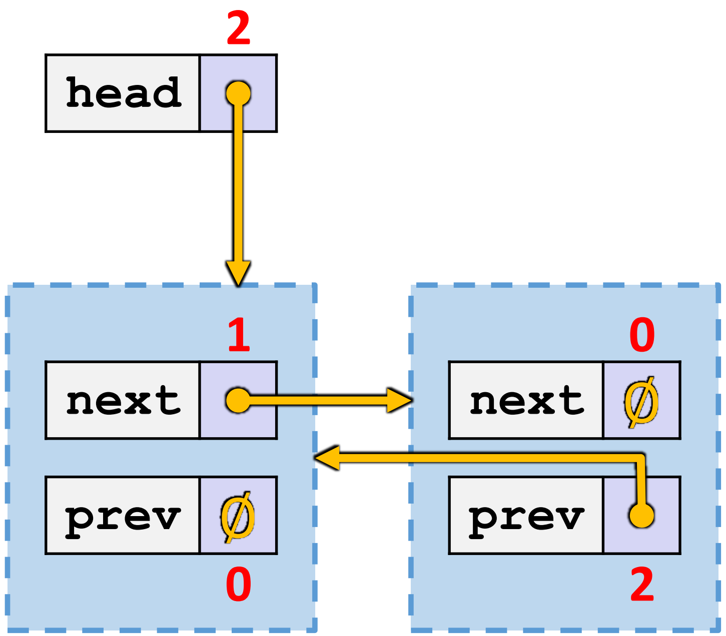
```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

struct A {
  shared_ptr<A> next;
  shared_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```



- What happens when we `delete head`?

std::weak_ptr

- `weak_ptr` is just like a `shared_ptr` but doesn't affect the reference count
  - Can *only* point to an object that is managed by a `shared_ptr`
  - Because it doesn't influence the reference count, `weak_ptr`s can become "*dangling*"
    - Object referenced may have been `delete`'d

- Can be used to break our cycle problem!

Breaking the Cycle with `weak_ptr`

weakcycle.cc

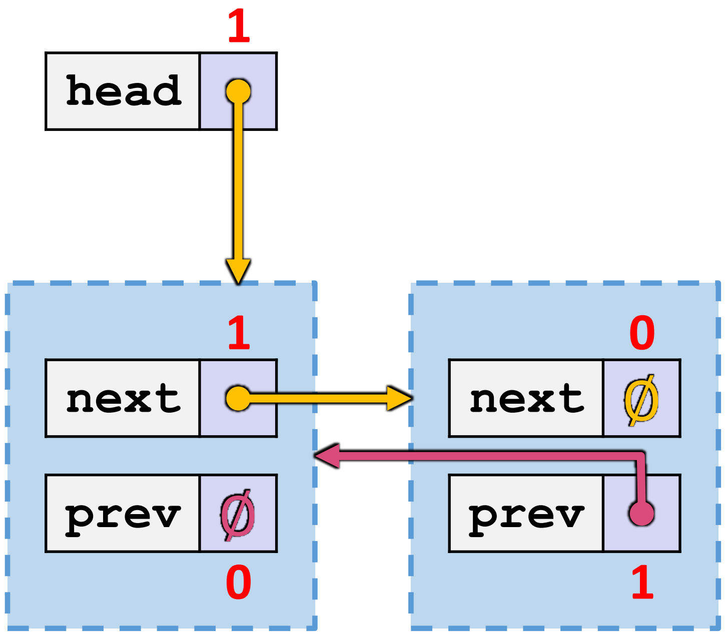
```
#include <cstdlib>
#include <memory>

using std::shared_ptr;
using std::weak_ptr;

struct A {
  shared_ptr<A> next;
  weak_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```



- Now what happens when we delete head?

Using a `weak_ptr`

usingweak.cc

```
#include <cstdlib>    // for EXIT_SUCCESS
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::shared_ptr, std::weak_ptr

int main(int argc, char **argv) {
  std::weak_ptr<int> w;

  {  // temporary inner scope
    std::shared_ptr<int> x;
    {  // temporary inner-inner scope
      std::shared_ptr<int> y(new int(10));
      w = y;
      x = w.lock();  // returns "promoted" shared_ptr
      std::cout << *x << std::endl;
    }
    std::cout << *x << std::endl;
  }
  std::shared_ptr<int> a = w.lock();
  std::cout << a << std::endl;

  return EXIT_SUCCESS;
}
```

Q

ANY QUESTIONS?