COMP 5421/1 BB Lecture notes August 12, 2018

1 Raw (Dumb) Pointers

- Major source of problems when pointing to dynamic memory
- Force you to manage deallocation of dynamic memory
- Force you to call `delete` at the right time and place exactly once
- Deallocating dynamic memory more than once cause a fatal run-time error
- Erasing a container's element that contains a dumb pointer does not `delete` that pointer; this can lead to a memory leak.
- If an exception occurs after a successful `new` statement but before the corresponding `delete` statement executes, a memory leak could occur.
- What's wrong with the following code?

```
void leakyTriple(int& x)
{
   int * pInt = new int(x);  // allocate storage

   if (isBad(x))             // validate x
      throw invalid_argument(std::to_string(x));

   x = *pInt * 3;            // triple x
   delete pInt;              // deallocate storage
   return;
}
```

If the `invalid_argument` exception is thrown, the `delete` statement isn't reached, and hence there is a memory leak.

However, as a local variables, `pInt` is removed from the stack memory; unfortunately, `pInt` is just an ordinary dumb pointer and *not a class object having a destructor*.

2 Smart Pointers

- Are NOT pointers
- Are objects that act like pointers
- Are objects that act smarter than dumb pointers!
- Help you mange your dynamically allocated memory (any resource)
- Help your program avoid memory leaks
- Smart pointer variables reside on the *stack*, not on the *heap*
- Stack variables are automatically destructed when they go out of scope

2.1 The easy parts

- Mimic regular pointers by implementing `operator->` and the unary `operator*`

```
template<typename T>
class SmartPointer
{
private:
    // ...
public:
    T* operator->() const;
    T& operator*() const;
    // ...
};
```

```
class Foo
{
public:
    void fun() { cout << "having Foo fun\n"; }
};
```

```
int main()
{
    SmartPointer<Foo> smart_foo(new Foo); // or new Foo()

    // smart_foo is NOT a pointer, but it acts like one:
    smart_foo->fun();
    (*smart_foo).fun();

    // notice there is no explicit call to delete!
    // smart_foo's dtor about to be called to do its thing
    return 0;
}
```

- Manage lifetime of objects pointed to by raw (dumb) pointers.

```
template<typename T>
class SmartPointer
{
private:
    // points to the object whose lifetime is
    // being managed by this smart pointer object
    T *rawPointer;
public:
    SmartPointer(T *p) : rawPointer(p) { } // ctor
    ~SmartPointer() { delete rawPointer; } // dtor
    T* operator->() const { return rawPointer; }
    T& operator*()  const { return *rawPointer;}
    // ...
};
```

```
int main()
{
    SmartPointer<int> sp(new int(10));
    std::cout << "before: " << *sp << endl;
    *sp = 15;
    std::cout << "after:  " << *sp << endl;

    // notice there is no explicit call to delete!
    // sp's dtor about to be called to do its thing
    return 0;
}
```

output

```
before: 10
after:  15
```

- What's wrong with the following code?

```
void triple(int& x)
{
    SmartPointer<int> smartIntPtr(new int(x));

    if (isBad(x))                // validate x
        throw invalid_argument(std::to_string(x));

    x = *smartIntPtr * 3;               // triple x

    return;
}
```

Nothing. Even if an exception is thrown, `smartIntPtr`'s destructor will still be called, which calls `delete` on the pointer it holds for you.

2.2 The difficult parts

- Define the copy constructor and copy assignment
  - Shallow copy
    - Two smart pointer objects sharing the same resource
    - Which object should manage the shared resource?
  - Deep copy
    - Two smart pointer objects, one of which is a copy of the other, but each manages its own resources
    - At what cost?
  - Institute the concept of exclusive ownership
    - Allow only one smart pointer to own a resource
      - Have copy assignment transfer ownership
      - Have destructor delete the resource
  - Institute the concept of shared ownership
    - Allow multiple smart pointers to share a resource, keeping track of their count
      - Have copy and assignment increment the count by 1
      - Have destructor decrement the count by 1 and delete the resource only when the count is zero
  - Other strategies?

3 C++ Smart Pointers

- auto_ptr
- unique_ptr
- shared_ptr
- weak_ptr

4 auto_ptr

- A C++98 attempt for a smart pointer; but, C++98 not ready!
- Does not work inside STL containers
- Deprecated in C++11
- Replaced with the unique_ptr class template in C++11
- Removed in C++17
- Implements the concept of exclusive ownership of dynamic memory
- When an auto_ptr variable goes out of scope, the underlying dynamic memory of that variable is freed by its destructor.
- Uses the transfer-of-ownership semantic during copy assignment operations

  Copying an auto_ptr sets its internal dumb pointer to nullptr

```
std::auto_ptr<Foo> ap_foo1(new Foo());
std::auto_ptr<Foo> ap_foo2;

ap_foo2 = ap_foo1;  // assignment operator
// transfer of memory ownership from ap_foo1 to ap_foo2
// ap_foo1's internal dumb pointer becomes nullptr
// ap_foo1->fun();   // CRASH ! ! !

ap_foo2->fun();  // ok

std::auto_ptr<Foo> ap_foo3(ap_foo2); // copy ctor
// transfer of memory ownership from ap_foo2 to ap_foo3
// ap_foo2's internal dumb pointer becomes nullptr
// ap_foo2->fun();   // CRASH ! ! !
```

- Replace all auto_ptr with unique_ptr, and never look back!

5 unique_ptr

- Should be your default smart pointer
- Introduced in C++11 to replace auto_ptr
- A unique_ptr exclusively "owns" the object to which it points
- Allows only one unique_ptr to point to a given object
- Forbids copy construction and assignment if the source unique_ptr object is NOT an rvalue :

```
unique_ptr<Foo> up_foo1(new Foo()); // up_foo1 is an lvalue
unique_ptr<Foo> up_foo2 (up_foo1); // Error: can't copy from an lvalue

unique_ptr<Foo> up_foo3;
up_foo3 = up_foo1;  // Error: can't assign from an lvalue
```

- Allows one unique_ptr to be copied/assigned to another unique_ptr only if the source unique_ptr is an rvalue

```
unique_ptr<Foo> up_foo1(new Foo());
unique_ptr<Foo> up_foo2(std::move(up_foo1)); //ok, source is an rvalue

unique_ptr<Foo> up_foo3;
up_foo3 = std::move(up_foo2); // ok, source is an rvalue
```

- std :: move() doesn't really move anything. It simply returns an rvalue reference to its argument, so that you can move data out of it, but you aren't necessarily required to do so.
- By default, when a unique_ptr goes out of scope, it calls `delete` on the pointer it holds to free its associated dynamic memory.

- Provides an ideal mechanism for returning dynamically allocated memory to client code.

```
// Returns a temporary unique_ptr
unique_ptr<double> make_double(double x)
{
   return unique_ptr<double>(new double(x));
}
```

```
int main()
{
   std::vector<unique_ptr<double>> vecUp(10);
   for (int i = 0; i < vecUp.size(); i++)
   {
      // assign a temporary unique_ptr to a unique_ptr in vecUp
      vecUp[i] = make_double(i * 1.0);
   }

   // ok,  passing a temporary unique_ptr as argument to push_back
   vecUp.push_back(make_double(15.0));

   // the for_each() statement below would fail
   // if the lambda were passed an object by value
   // instead of by reference because then it would
   // be necessary to copy a non-temporary unique_ptr
   // from vecUp to upx, which isn't allowed
   for_each(vecUp.begin(), vecUp.end(),
      [](unique_ptr<double> &upx) {cout << *upx << " "; });

   return 0;
}
```

- To use a `unique_ptr` to manage a dynamic array, simply include a pair of empty square brackets after the array base type:

```
// upArray below manages a dynamic array of ten uninitialized ints
unique_ptr<int[]> upArray(new int[10]);

// assign a value to each of the array elements
for (size_t k = 0; k < 10; ++k)
    upArray[k] = rand() % 100;

// when upArray goes out of scope, its destructor deletes the array
```

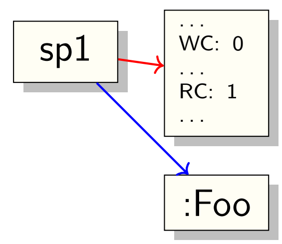
- You can always copy and/or assign a `unique_ptr` to `shared_ptr`, provided that the `unique_ptr` is an rvalue:

```
unique_ptr<Foo> up_foo1(new Foo());
shared_ptr<Foo> sp_foo1 = std::move(up_foo1); // ok, rhs is an rvalue
```

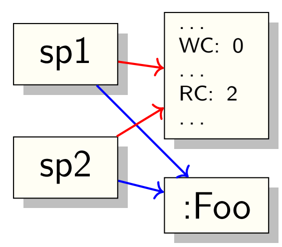
6 shared_ptr

- Stores a pointer to a dynamically allocated resource like memory that may shared with any number of other `shared_ptr` objects
- The internal pointer is `delete`ed only when the last `shared_ptr` object sharing the resource is destroyed.
- Uses *reference counting* to keep track of how many `shared_ptr` objects point to the resource:[1]

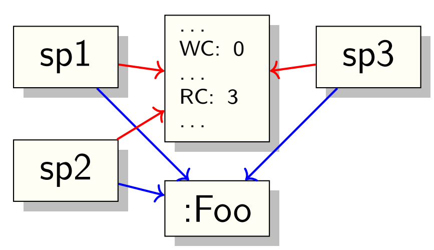
```
std::shared_ptr<Foo> sp1(new Foo());//ctor
```



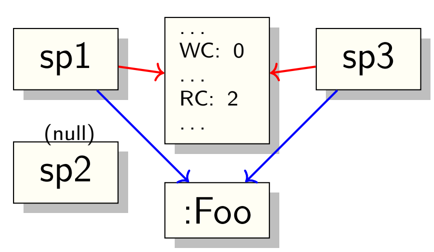

```
std::shared_ptr<Foo> sp1(new Foo()), sp2;
sp2 = sp1; // copy assignment
```




```
std::shared_ptr<Foo> sp1(new Foo()), sp2;
sp2 = sp1; // copy assignment
std::shared_ptr<Foo> sp3(sp1); // copy ctor
```

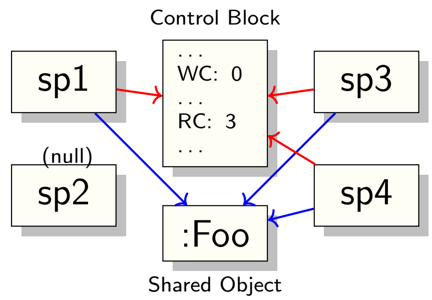



```
std::shared_ptr<Foo> sp1(new Foo()), sp2;
sp2 = sp1; // copy assignment
std::shared_ptr<Foo> sp3(sp1); // copy ctor
sp2 = nullptr; // copy assignment
```

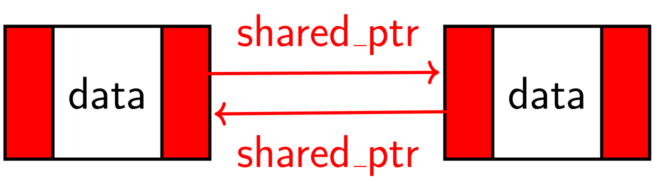


[1] WC stands for week count and RC for reference count

```
std::shared_ptr<Foo> sp1(new Foo()), sp2;
sp2 = sp1; // copy assignment
std::shared_ptr<Foo> sp3(sp1); // copy ctor
sp2 = nullptr; // copy assignment
auto sp4 = sp3; // copy ctor
```



- Can safely be copied and used in STL containers.
- By default, a `shared_ptr` object uses `delete` to free its associated dynamic resource; otherwise, it lets you supply a custom `deleter` function of your own.
- Consider a data structure such as a doubly linked list, where the shared pointers in two separate nodes each point at the other node:

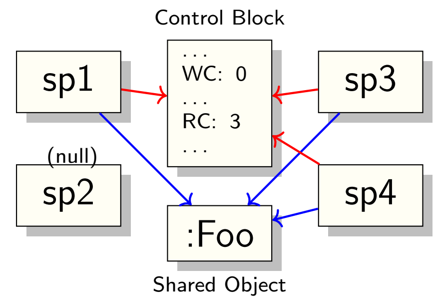


  As you can see our shared pointers form a circular reference, creating a case where the reference counts of the shared pointer objects each never drop to zero; hence, the resulting memory leak.

  Solution: use `weak_ptr` as suggested in the next section.

- C++11 offers the `make_shared` method that creates `shared_ptr` objects more efficiently than that of the `shared_ptr` constructor. (`make_shared` performs one heap-allocation, whereas `shared_ptr`'s constructor performs two.)

```
auto  sp1 = std::make_shared<Foo>();
auto  sp2 = sp1;
auto  sp3 = sp1;
sp2 = nullptr;
auto  sp4 = sp3;
```



7 weak_ptr

- A smart pointer that points to an object that is managed by a shared_ptr but does not control the lifetime of the object.
- Can bind to a shared_ptr without affecting the reference count of that shared_ptr – hence the name weak_ptr.
- As usual, once the last shared_ptr pointing to the object goes out of scope, the object itself will be deleted, regardless of whether or not there are weak_ptrs pointing to it.
- Can be converted to a shared_ptr using the shared_ptr constructor or the member function lock. That's how a weak_ptr can use the object it points to:

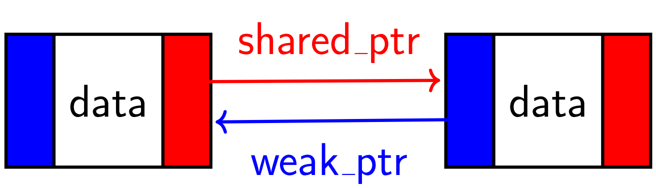
```
auto sp = std::make_shared<int>(13);

std::weak_ptr<int> wp(sp);

// ...

if (auto result = wp.lock())//result is of type std::make_shared<int>
{
    *result = 55;
}
cout << *sp << endl; // prints 55
```

- Solves the circular reference problem caused by the shared pointers in two separate objects each pointing at the other object:



Nodes in a doubly linked list

```
template < typename T >
class List {
   struct Node {
      T data;
      shared_ptr < Node > next; // shared_ptr to move forward
      weak_ptr < Node > prev; // weak_ptr to move backward
      Node():data(),next(), prev(){}
      Node( T x ):data( x ), next(), prev(){}
   };
shared_ptr < Node > front;
shared_ptr < Node > back;

public :
// ...
// ...
};
```

8 Rvalue References + Move Semantics

- An `lvalue` is an expression for which the program can obtain an address; for example, a variable name, a dereferenced pointer, a function that return a reference, etc.

- An `rvalue` is an expression to which one cannot apply the address operator; for example, 49 is an rvalue because the expression `&49` is not allowed; for another example, assuming `x` is an `int` variable, the expression `x+1` is an rvalue because `&(x+1)` is not allowed.

- Using the symbols `&&`, C++11 introduces `rvalue references` that can bind to `rvalues`:

```
int x = 10, &y = x;
int&& rv1 = 49;    // ok
//int&& rv2 = x;   // error: &x is allowed
//int&& rv2 = rv1; // error: &rv1 is allowed
int&& rv2 = x + 1; // ok
double && rv3 = std::sqrt(4.0); // ok
```

- Consider the following code:

```
std::string toLowerCase(const std::string& str)
{
   std::string temp = str;
   std::transform(temp.begin(), temp.end(), // source
      temp.begin(),  // destination
      [](char ch) {return std::tolower(ch); }); // lambda
   return temp;
}
```

```
9  int main()
10 {
11     cout << "Enter a very ... very long string of characters:\n";
12     std::string msg;
13     std::istream_iterator<char> start(std::cin), finish;
14     std::copy(start, finish, back_inserter(msg));
15 
16     std::string temp1(msg);  // make temp1 a copy of msg
17     std::string temp2(toLower(msg));//make temp2 a copy of a temporary
18     cout << temp1 << " " << temp2 << endl;
19     return 0;
20 }
```

- Note that on line 16 `std::string`'s ctor takes an `lvalue` as argument
- Note that on line 17 `std::string`'s ctor takes an `rvalue` as argument
- Wouldn't it be more efficient if `std::string`'s ctor knew that it could safely modify (steal resources of) its `rvalue` argument on line 17?
- In fact, it does. `std::string` does differentiate between the `lvalue` and `rvalue` arguments it receives on lines 16-17:

std::string move/copy constructors and assignment operators

```
string (const string& str);// copy ctor, called in line 16
string (string&& str) noexcept;// move ctor, called in line 17

string& operator= (const string& str); // copy assignment
string& operator= (string&& str) noexcept;// move assignment
```

See std::string::string and std::string::operator=. Note that the `rvalue` parameters are non-const (why?)

- Another example:

```
1  using std::string;
2  string make_copy(string& str); // matches non-const lvalue
3  string make_copy(const string& str); // matches const lvalue
4  string make_copy(string&& str); // matches rvalue
5  string toLower(const string& str);
6
7  int main()
8  {
9
10     cout << "Enter a very ... very long string of characters:\n";
11     string msg;
12     std::istream_iterator<char> start(std::cin), finish;
13     copy(start, finish, back_inserter(msg));
14
15     const string temp1 = make_copy(msg); // calls make_copy line 2
16     string temp2 = make_copy(temp1); // calls make_copy line 3
17     string temp3 = make_copy(toLower(msg));// calls make_copy line 4
18
19     cout << temp1 << " " << temp2 << " " << temp3 << endl;
20     return 0;
21 }
```