

Week 12 : Generic Algorithms, Associative
Containers and Dynamic Memory
Scientific Computing

Blair Jamieson

University of Winnipeg

Class 12

Outline

Algorithms and maps

Dynamic memory

Final exam format and topics

Outline

Algorithms and maps

- Standard library algorithms

- Introduction to templates

- `find_if` algorithm

- Function objects

- Numerical algorithms

- `map`

Dynamic memory

Final exam format and topics

Introduction to generic algorithms ¹

- ▶ About 80 algorithms – all useful for someone sometimes
- ▶ Can be found in `algorithm` header file
- ▶ Focus on ones with most usefulness:

<code>r=find(b,e,v)</code>	<code>r</code> points to first occur of <code>v</code> in <code>[b:e)</code>
<code>r=find_if(b,e,p)</code>	<code>r</code> points to first elem <code>x</code> in <code>[b:e)</code> such than <code>p(x)</code> is <code>true</code>
<code>x=count(b,e,v)</code>	<code>x</code> is num occur of <code>v</code> in <code>[b:e)</code>
<code>x=count_if(b,e,p)</code>	<code>x</code> is num elements in <code>[b:e)</code> such that <code>p(x)</code> is <code>true</code>
<code>sort(b,e)</code>	sort <code>[b:e)</code> using <code><</code> operator
<code>sort(b,e,p)</code>	sort <code>[b:e)</code> using <code>p</code>
<code>copy(b,e,b2)</code>	copy <code>[b:e)</code> to <code>[b2:b2+(e-b))</code> better be enough elems after <code>b2</code>

¹Notes based on Ch. 21 of B.Stroustrup, “Programming principles and practice using C++ ,” second edition, Addison Wesley, 2014.

Standard library algorithms

- ▶ List of useful algorithms continued from prev. page

<code>unique_copy(b,e,b2)</code>	copy <code>[b:e)</code> to <code>[b2:b2+(e-b))</code> don't copy adjacent duplicates
<code>merge(b,e,b2,e2,r)</code>	merge sorted sequences <code>[b2:e2)</code> and <code>[b:e)</code> into <code>[r:r+(e-b)+(e2-b2))</code>
<code>r=equal_range(b,e,v)</code>	<code>r</code> is sub-sequence of sorted range <code>[b:e)</code> with value <code>v</code>
<code>equal(b,e,b2)</code>	true if all elem of <code>[b:e) == [b2:b2+(e-b))</code>
<code>x=accumulate(b,e,i)</code>	<code>x</code> is sum of <code>i</code> and elements of <code>[b:e)</code>
<code>x=accumulate(b,e,i,op)</code>	as above, but use <code>op</code> instead of sum
<code>x=inner_product(b,e,b2,i)</code>	<code>x</code> is inner product of <code>[b:e)</code> with <code>[b2:b2+(e-b))</code>
<code>x=inner_product(b,e,b2,i,o,o2)</code>	use <code>o</code> and <code>o2</code> instead of <code>+</code> and <code>*</code>

Simplest algorithm : find

- ▶ `find` like most of the `algorithms` is a template function – which means it can be used with any of the container types
- ▶ We could use `find` without knowing details above, but lets have a closer look as it illustrates some useful design ideas
- ▶ The code for the templated function looks like:

```
1 template<typename In, typename T>
2 In find( In first, In last, const T& val ) {
3     while ( first != last && *first!=val ) ++first;
4     return first;
5 }
```

- ▶ `find` operates on a sequence defined by a pair of iterators
- ▶ Looks for value `val` in the sequence `[first:last)`
- ▶ If value is found it returns iterator to it, if it is not found returns iterator to one past end of sequence

Example use of `find`

- ▶ Lets consider searching for the index in a vector of an element with a particular value `val`:

```
void f( vector<int> & v, int x ) {  
    // below use auto instead of  
    // vector<int>::iterator  
    auto p = find( v.begin(), v.end(), x );  
    if ( p != v.end() ){  
        // we found x in v at iterator p  
        // index of value is:  
        auto idx = p-v.begin();  
    } else {  
        // no x found in v  
    }  
    // ...  
}
```

A note about the loop in `find`

- ▶ Note the form for the body of the `find` algorithm:

```
1 while ( first!=last && *first != val ) ++first ;  
2 return first ;
```

- ▶ The loop is not obvious – but is minimal, efficient and a direct rep of algorithm
- ▶ Pedestrian version of above might look like:

```
1 for (In p = first; p != last; ++p ) {  
2     if ( *p == val ) {  
3         return p;  
4     }  
5     return last ;  
6 }
```

- ▶ Difference is `for` loop had to allocate extra variable `p`, so the `while` loop is more efficient
- ▶ Also `while` loop has become popular style – so you will see it in other people's code

Some brief notes on templates

- ▶ Template is a mechanism that allows us to use types as parameters for a class or function
- ▶ Compiler generates specific class or function when we later provide the specific types as arguments
- ▶ The C++ notation for introducing a type parameter T is the `template<typename T>` prefix – meaning for all types T
- ▶ For example in defining the `find` function we used two `typename`s:

```
1 template<typename In, typename T>  
2 In find( In first, In last, const T& val );
```

- ▶ Note that when we used the `find` function we did not specify the template types – which works if the compiler can determine the types from the values passed to the function.

Again about using `find`

- ▶ The following functions are equivalent:

```
1 // function as before:
2 void f( vector<int> & v, int x ) {
3     auto p = find( v.begin(), v.end(), x );
4     if ( p != v.end() ){
5         //... use p
6     }
7 }
8 void g( vector<int> & v, int x ){
9     auto q = find< vector<int>::iterator, int >(
10        v.begin(), v.end(), x );
11    if ( q != v.end() ){
12        //... use q
13    }
14 }
```

Specialization

- ▶ To use a templated class we need to specify the type, for example when using vector:

```
1 vector<double> d; // vector of double
2 vector<char> c; // vector of characters
```

- ▶ We generate a *specialization* of vector that generates a new template instantiation
- ▶ Template instantiation takes place at compile time or link time – not run time
- ▶ Lets consider the vector interface on next slide!

vector template interface

```
1 template< typename T >
2 class vector {
3     int sz; // the size
4     T* elem; // pointer to elements
5     int space; // size + free space
6 public:
7     vector(): sz{0}, elem{nullptr}, space{0}{}
8     explicit vector(int s);
9     vector(const vector&); // copy constructor
10    vector& operator=(const vector&); // copy assign
11    ~vector(){ delete [] elem; } // destructor
12    T& operator[](int n) const {
13        return elem[n]; }
14    const T& operator[](int n) const {
15        return elem[n]; }
16    int size() const { return sz; }
17    int capacity() const { return space; }
18    void push_back( const T& d );
19    // ...
20 };
```

Notes vector template

- ▶ When we make `vector<double> d`, compiler generates all the member functions, for example:

```
// push_back for vector<double>:  
void vector<double>::push_back( const double & d  
    ){ /* ... */ }  
// Above was generated from template definition:  
template<typename T>  
void vector<T>::push_back( const T& d ){ /* ... */  
    }
```

- ▶ Note that instead of writing `template<typename T>` can also use `template<class T>` – they are synonymous

Generic programming

- ▶ Simplest definition: generic programming in C++ is using templates
- ▶ Better definition: “Writing code that works with a variety of types presented as arguments”
- ▶ A comparison of generic and object oriented programming:
 - ▶ *Generic programming* supported by templates, relies on compile-time type resolution
 - ▶ *Object-oriented programming* supported by class hierarchies and virtual functions, relies on run-time resolution
- ▶ Combinations of the two are possible and useful, for example:

```
1 void draw_lines( vector<Shape*>& v ) {  
2     for ( int i = 0; i<v.size(); ++i ) v[i]->draw();  
3 }
```

- ▶ Above uses templated vector, and vector of Shape* uses polymorphic objects that could be Line, Function, ...

Template summary

- ▶ Templates have many useful properties such as providing flexible types and good performance
- ▶ Some problems with templates:
 - ▶ poor separation between definition and interface
 - ▶ often spectacularly poor error diagnostics and messages
 - ▶ sometimes error messages come much later in compilation than location of error
- ▶ When compiling a use of template, compiler needs to look at the template arguments used to generate the code
- ▶ That includes all of member functions and all template functions
- ▶ For that reason template writers tend to put template definitions in header files
- ▶ When writing templates, start with specific type, then replace it with templated parameters, and test again with other types

Requirements from templated types

- ▶ C++ 14 provides mechanism to improve checking of template interfaces
- ▶ This is not implemented in all compilers yet though
- ▶ The mechanism is to put a set of requirements on the template arguments – this is called a *concept*
- ▶ For example `vector` requires that its elements can be copied or moved, can have their address taken, and be default constructed
- ▶ In C++ 14 the above set of requirements gets called an **Element**
- ▶ In C++ 14 the template definition for `vector` is:

```
1 template <typename T> // for all types T
2   requires Element<T>() // such that T is an
   element
3 class vector { // ... };
```

- ▶ Or in shorthand notation:

```
1 // for all types T such that Element<T>() is true
2 template <Element T>
3 class vector { // ... };
```


Requirements from templated types

- ▶ If compiler doesn't have C++ 14 support, specify requirements in comments:

```
1 template <typename Elem> // requires
   Element<Elem>()
2 class vector{ // ... };
```

- ▶ In that vein, the definition of the `find` algorithm, with requirements commented becomes:

```
1 template<typename In , typename T>
2 // requires Input_iterator<In>() &&
3 //           Equality_comparable< Value_type<T>>()
4 In find( In first , In last , const T& val ) {
5     while ( first != last && *first!=val ) ++first;
6     return first;
7 }
```

requires statements in C++ 14

- ▶ `Element<E>()` : `E` can be an element in a container
- ▶ `Container<C>()` : `C` can hold `Elements` and be accessed as a `[begin():end())` sequence
- ▶ `Forward_iterator<For>()` : `For` is an iterator that can be used to traverse a sequence `[b:e)`
- ▶ `Input_iterator<In>()` : `In` can be used to read a sequence `[b:e)` only once – like input stream
- ▶ `Output_iterator<Out>()` : `Out` A sequence that can be output using `Out`
- ▶ `Random_access_iterator<Ran>()` : `Ran` can be used to read and write a sequence `[b:e)` repeatedly and supports sub-scripting using `[]`
- ▶ `Allocator<A>()` : `A` can be used to acquire and release memory
- ▶ `Equal_comparable<T>()` : Can compare two `Ts` for equality using `==`
- ▶ `Equal_comparable<T,U>()` : Can compare `T == U` and get boolean result

requires statements in C++ 14 continued

- ▶ `Predicate<P,T>()` : We can call P with argument type T and get a boolean result
- ▶ `Binary_predicate<P,T>()` : We can call P with two arguments of type T and get boolean result
- ▶ `Binary_predicate<P,T,U>()` : We can call P with argument types T, U and get boolean result
- ▶ `Less_comparable<L,T>()` : We can use L to compare two Ts for less than and get boolean result
- ▶ `Less_comparable<L,T,U>()` : We can use L to compare a T with a U and get boolean result
- ▶ `Binary_operation<B,T>()` : We can use B to do an operation on two Ts
- ▶ `Binary_operation<B,T,U>()` : We can use B to do an operation using a T and a U
- ▶ `Number<N>()` : N behaves like a number and supports +, -, *, /

Containers and inheritance

- ▶ Note that we can't use a container of objects of derived class as container of objects of base class
- ▶ For example the following is not allowed:

```
1 vector <Shape> vs;  
2 vector <Circle> vc;  
3 vs = vc; // error: vector<Shape> required  
4 void ( vector<Shape>& );  
5 f( vc ); // error: vector<Shape> required
```

- ▶ Why not? We can:
 - ▶ Can convert `Circle*` to `Shape*`
 - ▶ Can convert `Circle&` to `Shape&`
- ▶ Reason: Not allowed to convert `Circle` to `Shape`, because it results in “slicing”

Containers and inheritance

- ▶ Okay, so how about this:

```
1 vector<Shape*> vps;  
2 vector<Circle*> vpc;  
3 vps=vpc; // error: vector<Shape*> required  
4 void f( vector<Shape*> & );  
5 f( vpc ); // error: vector<Shape*> required
```

- ▶ So, why aren't above okay? We are allowed to convert `Rectangle*` into a `Shape*`
- ▶ We can put a `Rectangle*` into a `vector<Shape*>`
- ▶ Reason this is not allowed:
 - ▶ Suppose `vector<Shape*>` was considered to be a `vector<Circle*>`
 - ▶ Would be in for nasty surprise if it came across a `Rectangle*`
- ▶ Inheritance is powerful, but templates do not implicitly extend its reach
- ▶ Just because D is a B, may not imply that `C<D>` is a `C`

find_if algorithm

- ▶ Often we are interested in finding a value if it meets some criteria (rather than just having an element that equals some value)
- ▶ `find_if` allows us to specify the criteria by supplying a function that returns a boolean (`true`) if the criteria are met
- ▶ The `find_if` algorithm is defined as:

```
1 template <typename In, typename Pred>
2 // requires Input_iterator<In>() &&
3 //           Predicate< Pred, Value_type<In>>()
4 In find_if( In first, In last, Pred( pred ) ){
5     while ( first != last && !pred( *first ) )
6         ++first;
7     return first;
8 }
```

- ▶ Looks similar to `find` algorithm, but instead of comparing `*first != val` it used `!pred(*first)`
- ▶ A predicate is a function that returns `true` or `false`

Example use of `find_if` algorithm

- ▶ Lets use `find_if` to find, in a container, the first element that is odd

```
1 bool odd( int x ){ return x%2; }
2 void f( vector<int>& v ) {
3     auto p = find_if( v.begin(), v.end(), odd );
4     if ( p!=v.end() ){
5         // found odd number ...
6     }
7 }
8 // note we could use above with any container type:
9 void g( list<int> & li ){
10     auto p = find_if( v.begin(), v.end(), odd );
11     if ( p!=v.end() ){
12         // found odd number ...
13     }
14 }
```

Example use of `find_if` algorithm

- ▶ Suppose we use `find_if` to find the first element, in a container, that is larger than 42:

```
1 bool larger_than_42( double x ){ return x>42; }
2 void f( list<double>& v ){
3     auto p = find_if( v.begin(), v.end(),
4         larger_than_42 );
5     if ( p!= v.end() ){ /* found value>42 ... */ }
}
```

- ▶ Above example is not too satisfying though
- ▶ What if we want to check if number is larger than 100, or any other arbitrary value?
- ▶ There must be a way of generalizing the function used in `find_if`

Function objects

- ▶ We would like to use `find_if` in the following way, so we can pass the value to compare to:

```
1 void f( list<double>& v, int x ){
2     auto p = find_if( v.begin(), v.end(),
3                     Larger_than(31) );
4     if ( p!=v.end() ){ /* found value > 31 ...*/ }
5     auto q = find_if( v.begin(), v.end(),
6                     Larger_than( x ) );
7     if ( q!=v.end() ){ /* found value > x ...*/ }
8 }
```

- ▶ In order for above to work we need `Larger_than` to:
 - ▶ Be called as a predicate, ie. `pred(*first)`
 - ▶ Be able to store a value such as `31` or `x`

A function object

- ▶ We define `Larger_than` as a function object in the following way:

```
1 class Larger_than {
2     int v;
3 public:
4     // constructor stores the argument
5     Larger_than( int vv ) : v (vv) {}
6     // operator() does comparison
7     bool operator()(int x) const { return x>v; }
8 };
```

- ▶ The notation `Larger_than(31)` constructs an object of type `Larger_than` that stores the value 31.
- ▶ When `find_if` tries to call `pred(*first)`, it calls the member function `operator()`
- ▶ In other words the member `operator()` is just the function call operator `()`, much the same as sub-scripting `v[i]` is given a meaning by `vector::operator[]`

Numerical algorithms

- ▶ There are only four numerical algorithms in the standard library, defined in `numeric` header file:
 1. `x = accumulate(b,e,i)` – Add a sequence of values stored in range of iterators `[b,e)` and add initial value `i` giving result `x`
 2. `x = inner_product(b,e,b2,i)` – Multiply pairs of values from two sequences and sum results. eg. for `[b,e)` holding `{a,b,c,d}` and `[b2,b2+(e-b))` holding `{e,f,g,h}`,
$$x = i + a * e + b * f + c * g + d * h$$
 3. `r = partial_sum(b, e, r)` – Produce sequence of sums of first `n` elements, eg. for `{a,b,c,d}` produce `{a, a+b, a+b+c, a+b+c+d}`
 4. `r=adjacent_difference(b,e,r)` – Produce sequence of differences between elements of sequence, eg. for `{a,b,c,d}` produces `{a,b-a,c-b,d-c}`

accumulate

- ▶ Simplest, and useful algorithm is `accumulate`, which adds a sequence of values:

```
1 template<typename In, typename T>
2 // requires Input_iterator<T> &&
3 //           Number<T>()
4 T accumulate( In first, In last, T init ) {
5     while( first != last ){
6         init = init + *first;
7         ++first;
8     }
9     return init;
10 }
```

- ▶ Initial value `init` is often referred to as the *accumulator*
- ▶ Every value in the sequence `[first:last)` is added to `init`

Example use of accumulate

- ▶ The type of the result (the sum) is the type of the variable that `accumulate()` uses, making it very flexible:

```
1 int a[]={1,2,3,4,5};
2 cout << accumulate(a,a+sizeof(a)/sizeof(int), 0 )
   <<end;
3 // above prints 15
4 void f( vector<double>& vd, int* p, int n){
5     double sum = accumulate( vd.begin(), vd.end(),
6                               0.0 );
7     int sum2 = accumulate( p, p+n, 0 );
8 }
9 void g( int *p, int n){
10    int s1 = accumulate( p, p+n, 0);
11    long s2 = accumulate( p, p+n, long{0} );
12    double s2 = accumulate( p, p+n, 0.0 );
13 }
```

- ▶ `long` was used to get more significant digits than `int`
- ▶ `double` can represent larger numbers than `int` but with possibly less precision

More example use of `accumulate`

- ▶ The variable used in initializer is used to specify the type being accumulated, but is passed by value:

```
1 void f( vector<double>& vd, int * p, int n ){
2     double s1 = 0.;
3     s1 = accumulate( vd.begin(), vd.end(), s1 );
4     // below is error, since can't pass s2 in same
5     // statement as it is defined
6     int s2 = accumulate( vd.begin(), vd.end(), s2 );
7     // below error is that we don't actually get
8     // result of accumulate:
9     float s3 = 0.;
10    accumulate( vd.begin(), vd.end(), s3 );
11 }
```

- ▶ `accumulate` can also be generalized by passing a fourth argument that is a binary operator – look it up if you are interested

map

- ▶ A `map` stores its keys in a way that makes it easy to see if a key is present, thus making the searching part trivial
- ▶ For example we may want to have a mapping between a list of people's names and a vector of people's phone numbers
- ▶ Here is an example where we keep a count of the times a given word is input:

```
1 map<string ,int> words; // keep (word,count) pairs
2 for ( string s; cin >> s; ) {
3     ++words[s];
4 }
5 for ( const auto & p : words ){
6     cout << p.first << " counted "
7         << p.second << " times " << endl;
8 }
```

Notes on map

- ▶ Our example made a map of (string, int) pairs
- ▶ That is, given a string, `words` can give us access to its corresponding `int`
- ▶ The line of code `++words[s]` does the following:
- ▶ Consider the word “sultan” in the string `s` that is not in the map yet:
 - ▶ Then “sultan” is entered into `words` with a default value for an `int` of zero
 - ▶ Now `words` will have an entry (“sultan”,0)
- ▶ the `++` before the `words[s]` increments the value making the pair (“sultan”,1)
- ▶ So the line of code `++words[s]` takes every word we get from the input and increases its value by one

Notes on map

- ▶ We can iterate through a map the same way as any standard library container
- ▶ The elements of a `map<string,int>` are of the type `pair<string,int>`
- ▶ Remember – the first member of a `pair` is called `first` and the second member `second`
- ▶ The output loop is then clear:

```
1 for ( const auto & p : words ) {  
2     cout << p.first << " counted "  
3         << p.second << " times " << endl;
```

Notes on `map` and `unordered_map`

- ▶ `map` is implemented as a binary tree
- ▶ `unordered_map` is implemented as a hash table
- ▶ To find an element in a `vector`, `find()` needs to examine all of the elements from the beginning to the element of right value, or the end
 - ▶ On average cost of running `find` is proportional to the length of the vector \mathcal{N} is $\mathcal{O}(\mathcal{N})$
- ▶ Element in `map` are stored in a binary tree – at each level in the tree there is a link to elements that are larger than and smaller, so for a tree of \mathcal{N} elements, average time to run `find` is $\mathcal{O}(\log_2 \mathcal{N})$
- ▶ A hash-table's `find` is $\mathcal{O}(1)$ – so it is obviously important when being used for large tables, such as a map of 500,000 web addresses
- ▶ Use of `unordered_map` is left as exercise for the student – it's use is similar to the other containers we have looked at
- ▶ `unordered_map` uses a hash function to calculate index to `key, pair` – beyond scope of this class

Lookup in vector, map and unordered_map

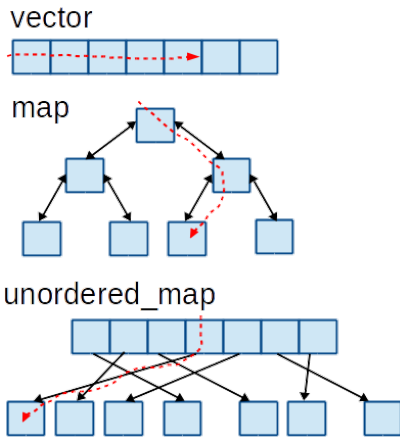


Figure: Looking up value in container

Notes on set

- ▶ **set** is a **map** without values
- ▶ In other words, a set is an ordered **set** of keys
- ▶ By default **set** will use **<** operator unless binary operation is supplied as second template argument
- ▶ For example we might have a structure holding information about fruit:

```
1 struct Fruit{
2     string name;
3     int count;
4     double unit_price;
5     Date last_sale_date; //...
6 };
7 struct Fruit_order{
8     bool operator()( const Fruit & a,
9                     const Fruit & b ) const {
10         return a.name() < b.name();
11     }
12 }
13 // below is set of Fruit objects that are sorted
14 // using the Fruit_order function object
15 set< Fruit , Fruit_order> inventory;
```

Outline

Algorithms and maps

Dynamic memory

- Dynamic memory and smart pointers

- Dynamic memory allocation

- Dynamic arrays

- Using the STL containers and dynamic memory

Final exam format and topics

Smart pointers ²

- ▶ So far the objects we have created were local, automatic objects that are destroyed when the block they are defined in comes to an end
- ▶ Static variables we used are created before their first use and are destroyed when the program ends
- ▶ C++ also has dynamically allocated objects that have a lifetime independent of their scope
- ▶ Properly freeing dynamically allocated objects can be source of memory leaks and surprising bugs
- ▶ To help manage dynamically allocated objects two smart pointer types exist: `shared_ptr` which allows multiple pointers to refer to same object, and `unique_ptr` which “owns” the object to which it points

²Notes on dynamic memory and smart pointers are based on Ch.12 of S.Lippman, J. Lajoie, B. Moo, “C++ Primer,” 5th edition.

Memory types

- ▶ Static memory is used for static objects
- ▶ Stack memory is used for non-static objects defined in functions
- ▶ Free store or heap memory is used for dynamically allocated objects
- ▶ C++ dynamic memory is allocated using the **new** operator, which returns a pointer to the allocated memory
- ▶ dynamic memory is freed using the **delete** operator which takes a pointer to the memory to be freed
- ▶ Two possible problems to be careful of:
 - ▶ Freeing the dynamic memory and still trying to use the object that used to be there
 - ▶ Forgetting to free the dynamic memory after it is no longer needed, resulting in a memory leak

shared_ptr class

- ▶ The first smart pointer to help with dynamic memory is the templated `shared_ptr` class
- ▶ The template type supplies the name of the kind of objects in the memory being pointed to
- ▶ Example declarations of `shared_ptr` objects:

```
1 // empty pointer that can point to a string
2 shared_ptr<string> p1;
3 // empty pointer that can point to a list of ints
4 shared_ptr< list<int> > p2;
5 // Use of shared_ptr is similar to use of pointers
6 // check if p1 is not null, then check if its
   string is empty
7 if ( p1 && p1->empty() ) *p1 = "hi";
```


Use of `make_shared` function

- ▶ Safest way to allocate dynamic memory is with the `make_shared` function
- ▶ Along with smart pointers they are defined in `memory` header file
- ▶ `make_shared` function is also templated, taking the type of object in the memory as the template type
- ▶ Function arguments are used as the constructor arguments to the object being allocated, eg:

```
1 // point to an int initialized to 42:
2 shared_ptr<int> p3 = make_shared<int>( 42 );
3 // point to a string with value 9999999999
4 shared_ptr<string> p4 = make_shared<string>( 10,
5     '9' );
6 // point to an int value initialized to zero
7 shared_ptr<int> p5 = make_shared<int>();
```

Copying and assigning `shared_ptr` objects

- ▶ Nice thing about smart pointers is that they keep track of how many `shared_ptrs` point to the same object, eg:

```
1 // object pointed to by p has one user:
2 auto p = make_shared<int>( 42 );
3 // now object has two users:
4 auto q( p ); // p and q point to same object
```

- ▶ If count of number of objects pointed to goes to zero, `smart_ptr` deletes the memory being pointed to, eg:

```
1 // int to which r points below now has one user (r)
2 auto r = make_shared<int>(42);
3 // assign r a new value... now nobody points to
4 // the previous memory holding the integer 42
5 // therefore that memory gets deleted.
6 r = q; // r now points to object pointed to by q
```

Table of operations common to `shared_ptr` and `unique_ptr`

<code>shared_ptr<T> sp</code>	Null smart pointer to point to
<code>unique_ptr<T> up</code>	objects of type T
<code>p</code>	below p is one of <code>sp</code> or <code>up</code>
<code>*p</code>	when tested <code>true</code> they point to object
<code>p->member</code>	dereference pointer to get
<code>p.get()</code>	value of object pointed to by pointer
<code>swap(p, q)</code>	synonymous with <code>(*p).member</code>
<code>p.swap(q)</code>	returns pointer in p
	swaps pointers in p and q

Table of operations specific to `shared_ptr`

<code>make_shared<T>(args)</code>	returns a <code>shared_ptr</code> that points to dynamic mem type <code>T</code> , <code>args</code> inits object
<code>shared_ptr<T> p(q)</code>	<code>p</code> is a copy of <code>q</code> , increments count in <code>q</code> . Pointer <code>q</code> must convert to <code>T*</code>
<code>p = q</code>	decrement <code>p</code> 's count, incr <code>q</code> 's count if count of <code>p</code> is zero delete <code>p</code> 's mem
<code>p.unique()</code>	returns <code>true</code> if count of <code>p</code> is one
<code>p.use_count()</code>	returns number of objects sharing <code>p</code>

Automatic destruction of `shared_ptr` objects

- ▶ Consider function returning `shared_ptr` to object of type `Foo`:

```
1 // function factory'' returns shared_ptr to
2 // dynamically allocated object
3 shared_ptr<Foo> factory ( T arg ){
4     // process arg as appropriate
5     // shared_ptr will take care of deleting memory
6     return make_shared< Foo >( arg );
7 }
```

- ▶ Since `factory` function returns a `shared_ptr` memory will be freed once the pointer goes out of scope, for example:

```
1 void use_factory( T arg ){
2     shared_ptr<Foo> p = factory( arg );
3     // use p for something
4 } // p goes out of scope
5 // memory p points to automatically freed
```

Automatic destruction of `shared_ptr` objects

- ▶ Note that if you store `shared_ptr` objects in a container the objects they point to will remain in memory
- ▶ If you no longer need the object pointed to by the `shared_ptr` make sure to remove the element from the container
- ▶ Program will still function if you don't but may take up extra memory

Classes with resources with dynamic lifetime

- ▶ Often use dynamic memory for one of three reasons:
 1. Don't know how many objects that you need
 2. Don't know precise type of objects needed
 3. Want to share data between several objects
- ▶ Container classes use dynamic memory internally to allow the number of elements in the container to change
- ▶ Lets consider objects that share the same underlying data – this is different than the container classes who own their data

```
1 vector<string> v1;  
2 { // new scope  
3     vector<string> v2 = {"one", "two", "three"};  
4     v1 = v2; // copy elements from v2 to v1  
5 } // v2 destroyed  
6 // v1 has three elements (copied from v2 before)
```

Shared data

- ▶ Consider a class `StrBlob` that keeps a copy of a pointer to a vector of data
- ▶ Copies of a `StrBlob` object share elements

```
1 StrBlob b1; // empty StrBlob
2 { // new scope
3   StrBlob b2 = {"one", "two", "three"};
4   b1 = b2; // b1 and b2 share same elements
5 } // b2 is destroyed, but elements in b2 must not
   be
6 // b1 points to elements originally created in b2
```

- ▶ When `b1` goes out of scope we don't want the elements to go away since `b2` refers to them

Defining the StrBlob class

```
1 class StrBlob {
2 public:
3     typedef vector<string>::size_type size_type;
4     StrBlob();
5     StrBlob( initializer_list<string> il);
6     size_type size() const { return data->size(); }
7     bool empty() const { return data->empty(); }
8     // add and remove elements
9     void push_back(const string &t) {
10         data->push_back(t); }
11     void pop_back();
12     string& front(); //
13     string& back();
14 private:
15     shared_ptr< vector<string> > data;
16     // throws msg if data[i] isn't valid
17     void check(size_type i, const std::string &msg)
18         const;
19 };
```

Notes on the StrBlob class

- ▶ Implemented `size`, `empty` and `push_back` by forwarding their work via the data pointer to the underlying vector
- ▶ For example `StrBlob::size()` calls `data->size()`
- ▶ Two constructors are defined – a default constructor that makes an empty vector, and an `initializer_list` one:

```
1 StrBlob::StrBlob() :  
    data(make_shared<vector<string>>()) { }  
2 StrBlob::StrBlob(initializer_list<string> il):  
3     data(make_shared<vector<string>>(il)) { }
```

- ▶ Both constructors use underlying vector initialization

StrBlob element access members

- ▶ `pop_back`, `front`, `back` operations access members in the vector
- ▶ All must check if element exists, therefore defined `private` function to do check:

```
1 void StrBlob::check(size_type i,
2                       const string &msg) const {
3     if (i >= data->size())
4         throw out_of_range(msg);
5 }
6 string& StrBlob::front() {
7     check(0, "front on empty StrBlob");
8     return data->front();
9 }
10 string& StrBlob::back() {
11     check(0, "back on empty StrBlob");
12     return data->back();
13 }
14 void StrBlob::pop_back() {
15     check(0, "pop_back on empty StrBlob");
16     data->pop_back();
17 } // above 3 should be overloaded on const
```

StrBlob copy, assign, destroy

- ▶ Uses default operations for copy, assign, destroy
- ▶ Only member data is `shared_ptr` which gets copied, assigned or destroyed
- ▶ So when a `StrBlob` is copied, the `shared_ptr` count gets incremented
- ▶ Also when it is assigned, the count is incremented
- ▶ When a `StrBlob` is destroyed, the `shared_ptr` count is decremented
- ▶ If the `shared_ptr` count is reduced to zero, the memory for the shared data is deleted

Managing memory directly

- ▶ Can directly use `new` to allocate memory and `delete` to free memory allocated by `new`
- ▶ Can be error prone, so care is needed – recommended to use smart pointers when possible
- ▶ `new` returns a pointer to the object it allocates:

```
1 // pi points to dynamically allocated un-named
2 // uninitialized int
3 int *pi = new int;
```

- ▶ `pi` points to memory allocated on free store to hold an `int`
- ▶ Values allocated on free store are default initialized (undefined value for built-in types, or class types are initialized by default constructor):

```
1 // below : initialized to empty string
2 string *ps = new string;
```

Managing memory directly – more examples using `new`

- ▶ Can also value initialize dynamically allocated object by following type name with pair of parentheses:

```
1 // default initialized to the empty string
2 string *ps1 = new string;
3 // value initialized to the empty string
4 string *ps = new string();
5 // default initialized; *pi1 is undefined
6 int *pi1 = new int;
7 // value initialized to 0; *pi2 is 0
8 int *pi2 = new int();
```

- ▶ It is a good idea to initialize dynamically allocated objects

Initializing values using `new`

- ▶ Can use `auto` to deduce the type of the object from that initializer
- ▶ Can only be done with a single initializer inside parentheses:

```
1 // p1 points to an object of the type of obj
2 // that object is initialized from obj
3 auto p1 = new auto(obj);
4 // error: must use parentheses for the initializer
5 auto p2 = new auto{a,b,c};
```

- ▶ Type of `p1` is a pointer to the auto-deduced type of `obj`
- ▶ If `obj` is an `int`, then `p1` is `int*`
- ▶ It is legal to use `new` to allocate `const` objects:

```
1 // allocate and initialize a const int
2 const int *pci = new const int(1024);
3 // allocate a default-initialized const empty
  string
4 const string *pcs = new const string;
```

Memory exhaustion

- ▶ In unfortunate cases computer's memory may all be in use (or we request to create too large an object for the memory)
- ▶ If `new` fails to allocate the requested storage, it throws an exception of type `bad_alloc`
- ▶ We can request `new` to just fail instead of throwing an exception:

```
1 // if allocation fails , new throws std::bad_alloc
2 int *p1 = new int ;
3 // if allocation fails , new returns a null pointer
4 int *p2 = new (nothrow) int ;
```

- ▶ Latter form of `new` is called **placement new**

Freeing dynamic memory

- ▶ Must return dynamically allocated memory to system when we are finished with it
- ▶ `delete` takes pointer to object we want to free:

```
1 delete p; // p must point to dynamic memory or null
```

- ▶ Deleting pointer to memory that was not allocated by `new`, or deleting same pointer value more than once is undefined:

```
1 int i, *pi1 = &i, *pi2 = nullptr;  
2 double *pd = new double(33), *pd2 = pd;  
3 delete i; // error: i not pointer  
4 delete pi1; // undefined: pi1 refers to a local  
5 delete pd; // ok  
6 delete pd2; // undefined: memory at pd2 already  
   freed  
7 delete pi2; // ok to delete null pointer
```

Freeing dynamic memory

- ▶ Note that const objects allocated dynamically can be deleted

```
1 const int *pci = new const int(1024);  
2 delete pci; // ok: deletes a const object
```

- ▶ Dynamically allocated objects exist until they are freed
- ▶ Memory of `shared_ptr` is automatically deleted when last `shared_ptr` is destroyed
- ▶ Dynamic object is managed by pointer – if you lose the pointer then you have a memory leak – you lose address to memory that needs to be cleaned up

Freeing dynamic memory

- ▶ Functions returning pointers to dynamic memory put burden on caller to delete the memory:

```
1 // factory returns a pointer to a dynamically
   // allocated object
2 Foo* factory(T arg) {
3     // process arg as appropriate
4     // caller is responsible for deleting this memory
5     return new Foo(arg);
6 }
```

- ▶ Caller may forget to clean up memory:

```
1 void use_factory(T arg) {
2     Foo *p = factory(arg);
3     // use p but do not delete it
4 }
5 // p goes out of scope,
6 // but the memory to which p points is not freed!
```

Freeing dynamic memory

- ▶ Dynamic memory managed through built-in pointers (rather than smart pointers) exists until it is explicitly freed.
- ▶ Better `use_factory` function:

```
1 void use_factory(T arg){
2     Foo *p = factory(arg);
3     // use p
4     // remember to free the memory
5     // now that we no longer need it
6     delete p;
7 }
```

Freeing dynamic memory

- ▶ After delete can set pointer to `nullptr`
- ▶ Doesn't guarantee that another object doesn't hold a pointer to that memory, for example:

```
1 // p points to dynamic memory
2 int *p(new int(42));
3 // p and q point to the same memory
4 auto q = p;
5 // invalidates both p and q
6 delete p;
7 // indicates that p is no longer bound to an object
8 p = nullptr;
```

- ▶ `q` still holds address of memory that is no longer allocated
- ▶ `q` is called a **dangling pointer** that refers to memory that once held an object, but no longer does

Common problems with `new` and `delete`

1. Forgetting to delete memory – results in memory leak that can be hard to detect until application is run for enough time to fill memory
2. Using object after it is deleted – can be detected by making pointer null after delete
3. Deleting same memory twice – can happen if two pointers address same object – result is corrupting free store
 - ▶ Above errors are easier to make than to find and fix
 - ▶ Smart pointers can be used to take care of deleting memory when there are no more users of the memory

Using `shared_ptr` with `new`

- ▶ Can initialize smart pointer from a pointer returned by `new`:

```
1 shared_ptr<double> p1; // shared_ptr that can
   point at a double
2 shared_ptr<int> p2(new int(42)); // p2 points to
   an int with value 42
3 shared_ptr<int> p3 = new int(1024); // error:
   must use direct
4 initialization
5 shared_ptr<int> p4( new int(1024) ); // ok: uses
   direct initialization
```

- ▶ Since implicit conversion of pointer to smart_pointer is not allowed, need to explicitly bind `shared_ptr` to pointer in a return value:

```
1 shared_ptr<int> clone(int p) {
2   // error: implicit convert to shared_ptr<int>
3   return new int(p);
4 }
5 shared_ptr<int> clone(int p) {
6   // ok: create shared_ptr<int> from int*
7   return shared_ptr<int>(new int(p));
8 }
```

Using `shared_ptr` with `new`

- ▶ A caution against using `new` and `shared_ptr`
- ▶ Consider a function taking a `shared_ptr` as argument:

```
1 // ptr is created and initialized when process is
   // called
2 void process(shared_ptr<int> ptr)
3 {
4     // use ptr
5 } // ptr goes out of scope and is destroyed
6 //...
7 // correct way to use process:
8 shared_ptr<int> p(new int(42));
9 // reference count is 1
10 // now copying p increments its count;
11 // in process the reference count is 2
12 process(p);
13 int i = *p; // ok: reference count is 1
```


Using `shared_ptr` with `new`

- ▶ Now consider what happens when using `process` function with memory allocated with `new`:

```
1 // dangerous: x is a plain pointer, not a smart
  pointer
2 int *x(new int(1024));
3 // error below: cannot convert int* to
  shared_ptr<int>
4 process(x);
5 // below is legal, but the memory will be deleted!
6 process(shared_ptr<int>(x));
7 int j = *x; // undefined: x is a dangling pointer!
```

More cautions for `shared_ptr`

- ▶ `shared_ptr` provides a `get` method which returns a bare pointer
- ▶ Compiler won't complain if you bind another smart pointer to pointer returned by `get`
- ▶ However, the two smart pointers aren't smart enough to know about each other:

```
1 shared_ptr<int> p(new int(42)); // reference count
   is 1
2 //below ok: but don't use q in any way that might
   delete its
3 int *q = p.get();
4 pointer
5 { // new block
6   // undefined: two independent shared_ptrs
7   // point to the same memory
8   shared_ptr<int>(q);
9 }
10 // block ends, q is destroyed,
11 // and the memory to which q points is freed
12 int foo = *p; // undefined; the memory to which p
```

Other shared_ptr operations

- ▶ We can use `reset` to assign a new pointer to `shared_ptr`

```
1 p = new int(1024);           // error: cannot assign a
    pointer to a shared_ptr
2 p.reset(new int(1024));     // ok: p points to a new
    object
```

- ▶ Updates reference count, and if count goes to zero deletes object to which p points
- ▶ Often used together with `unique` method to change object shared among several `shared_ptr` objects:

```
1 if (!p.unique()) // we aren't alone; alloc new copy
2     p.reset(new string(*p));
3 // now that we know we're the only pointer,
4 // okay to change this object
5 *p += newVal;
```

Smart pointers and dumb classes

- ▶ Many C++ classes, including library classes, define destructors to clean up resources used by object
- ▶ For classes that do not define destructors may have them forget to release resources – particularly if an exception happens
- ▶ For example imagine network library use:

```
1 struct destination; // what we connect to
2 struct connection; // info for connection
3 connection connect(destination*); // open connect.
4 void disconnect(connection); // close connection
5 void f(destination &d /* other parameters */) {
6     // get a connection; must remember to
7     // close it when done
8     connection c = connect(&d);
9     // use the connection
10    // if we forget disconnect before exiting f,
11    // there will be no way to close c
12 }
```

- ▶ If `connection` had a destructor, it would automatically close connection
- ▶ If `connection` has no destructor, we can use `shared_ptr`

Smart pointers to add deletion to dumb classes

- ▶ By default `shared_ptr` assumes they point to dynamic memory
- ▶ When `shared_ptr` is destroyed it executes `delete` on pointer it holds
- ▶ To use `shared_ptr` to manage `connection` must define function to use in place of `delete`
- ▶ In this case deleter must take a single argument of type `connection *`:

```
1 void end_connection(connection *p) {  
2     disconnect(*p);  
3 }
```

Smart pointers to add deletion to dumb classes

- ▶ Our function to do the connection now looks like:

```
1 void f(destination &d /* other parameters */) {  
2     connection c = connect(&d);  
3     shared_ptr<connection> p(&c, end_connection);  
4     // use the connection  
5     // when f exits, even if by an exception,  
6     // the connection will be properly closed  
7 }
```

- ▶ When `p` is destroyed, it doesn't delete its stored pointer, instead it calls `end_connection` on that pointer
- ▶ This ensures connection is closed even if function exits abnormally

Smart pointer cautions

- ▶ Smart pointers make use of dynamic memory less error prone, however there are a few pitfalls to avoid:
 - ▶ Don't use same built-in pointer value to initialize or **reset** more than one smart pointer
 - ▶ Don't **delete** the pointer returned from `get()`
 - ▶ Don't use `get()` to initialize or **reset** another smart pointer
 - ▶ If you use smart pointer to manage resource other than memory, remember to pass a deleter

unique_ptr operations

- ▶ `unique_ptr` owns the object it points to – only one at a time can point to a given object

<code>unique_ptr<T> u1</code>	null pointer to object of type T u1 uses delete to free pointer
<code>unique_ptr<T,D> u2</code>	uses object type D to free pointer
<code>unique_ptr<T,D> u(d)</code>	uses d of type D in place of delete
<code>u = nullptr</code>	deletes object u points to, makes u null
<code>u.release()</code>	release control of ptr u had, makes u null
<code>u.reset()</code>	deletes object to which u points
<code>u.reset(q)</code>	makes u point to q
<code>u.reset(nullptr)</code>	makes u null

unique_ptr example use

- ▶ `unique_ptr` doesn't have a function to make a new `unique_ptr`
- ▶ Example use:

```
1 // unique_ptr that can point at a double
2 unique_ptr<double> p1;
3 // p2 points to int with value 42
4 unique_ptr<int> p2( new int(42) );
5 unique_ptr<string> p3( new string( "Stegosaurus" ) );
6 // error: no copy for unique_ptr
7 unique_ptr<string> p4(p3);
8 unique_ptr<string> p5;
9 // error: no assign for unique_ptr
10 p5 = p4;
```

unique_ptr example use

- ▶ While we can't assign or copy, we can transfer ownership of `unique_ptr` by calling `release` or `reset`:

```
1 unique_ptr<string> p1(new string("Stegosaurus"));
2 // transfer ownership from p1 (which points to
3 // the string Stegosaurus) to p2
4 // release makes p1 null
5 unique_ptr<string> p2(p1.release());
6 unique_ptr<string> p3(new string("Trex"));
7 // transfer ownership from p3 to p2
8 // reset deletes the memory to which p2 had
9 p2.reset(p3.release());
10 // below WRONG: p2 won't free memory and lose
    pointer
11 p2.release();
12 // below ok, but we must remember to delete(p)
13 auto p = p2.release();
```

- ▶ Note that `release` returns pointer and makes it null, so it is now up to you to delete the memory at that pointer

passing and returning `unique_ptr`

- ▶ We can only copy or assign `unique_ptr` when it is about to be destroyed
- ▶ Commonly done when return from function

```
1 unique_ptr<int> clone( int p ){
2     // ok: explicitly create from int*
3     return unique_ptr<int>( new int(p) );
4 }
5 // also ok to return local copy of unique_ptr:
6 unique_ptr<int> clone( int p ){
7     unique_ptr<int> ret( new int(p) );
8     // ...
9     return ret;
10 }
```

Overriding unique_ptr deleter

- ▶ Also need to supply type of deleter inside angle brackets
- ▶ In our previous network connection example we could use:

```
1 void f(destination &d /* other params... */){
2     connection c = connect(&d); // open the
        connection
3     // when p is destroyed, the connection will be
        closed
4     unique_ptr<connection, decltype(end_connection)*>
        p(&c, end_connection);
5
6     // use the connection
7     // when f exits, even if by an exception,
8     // the connection will be properly closed
9 }
```

weak_ptr

- ▶ `weak_ptr` is a smart pointer that does not control lifetime of memory it points to
- ▶ Can bind a `weak_ptr` to a `shared_ptr` and not affect the reference count in `shared_ptr`
- ▶ `weak_ptr` operations:

<code>weak_ptr<T> w</code>	Null pointer to type T objects
<code>weak_ptr<T> w(sp)</code>	Pointer to sp (same as type T) objects
<code>w=p</code>	w now points to p p is <code>weak_ptr</code> or <code>shared_ptr</code>
<code>w.reset()</code>	makes w null
<code>w.use_count()</code>	num <code>shared_ptr</code> s with ownership of w
<code>w.expired()</code>	returns true if <code>use_count</code> is zero
<code>w.lock()</code>	if <code>expired</code> true, returns null, otherwise return <code>shared_ptr</code> w points at

Making a `weak_ptr`

- ▶ We create a `weak_ptr` from a `shared_ptr`:

```
1 auto p = make_shared<int>(42);  
2 weak_ptr<int> wp(p);  
3 // wp weakly shares with p;  
4 // use count in p is unchanged
```

- ▶ It is possible object `wp` points to will be deleted
- ▶ To access object must call `lock` which checks if object still exists – if so returns `shared_ptr` – if not returns `nullptr`

Example use of `weak_ptr`

- ▶ Define a companion pointer to `StrBlob` class that stores a `weak_ptr` to the data in a `StrBlob`
- ▶ Call our pointer class `StrBlobPtr`
- ▶ By using `weak_ptr` don't affect lifetime of data, but can prevent user from accessing vector that doesn't exist

StrBlobPtr

```
1 // StrBlobPtr throws an exception on attempts
2 // to access a nonexistent element
3 class StrBlobPtr {
4 public:
5     StrBlobPtr(): curr(0) { }
6     StrBlobPtr(StrBlob &a, size_t sz = 0):
7         wptr(a.data), curr(sz) { }
8     string& operator*() const; // dereference
9     StrBlobPtr& operator++(); // prefix version
10 private:
11     // check returns a shared_ptr to the vector
12     // if the check succeeds
13     shared_ptr< vector<string> >
14         check( size_t, const string&) const;
15     // store a weak_ptr, which means the underlying
16     // vector might be destroyed
17     weak_ptr< vector<std::string> > wptr;
18     size_t curr; // current position within the array
19 };
```


Notes on StrBlobPtr

- ▶ Default constructor generates a null `StrBlobPtr` with `wptr` set to null and `curr` set to zero
- ▶ Initializer list constructor `wptr` to point to `vector` in shared ptr given to constructor and `curr` to `sz`
- ▶ `check` member function needs to check existence of vector:

```
1 shared_ptr< vector<string> >  
    StrBlobPtr::check(size_t i, const string &msg)  
    const {  
2     auto ret = wptr.lock(); // does vector exist?  
3     if (!ret)  
4         throw runtime_error("unbound StrBlobPtr");  
5     if (i >= ret->size()) throw out_of_range(msg);  
6     return ret; // return shared_ptr to vector  
7 }
```

Notes on StrBlobPtr

- ▶ Dereference operator overloads operator*():

```
1 std::string& StrBlobPtr::operator*() const {  
2     auto p = check(curr, "dereference past end");  
3     return (*p)[curr];  
4     // (*p) is the vector to which this object points  
5 }
```

- ▶ If check succeeds, p is a shared ptr to the vector
- ▶ (*p)[curr] dereferences that shared ptr to get the vector, and [curr] gets the element in the vector at curr

Notes on StrBlobPtr

- ▶ Increment operator overloads `operator++()` to increment and return reference to incremented object

```
1 StrBlobPtr& StrBlobPtr::operator++() const {  
2     check(curr, "increment past end of StrBlobPtr");  
3     ++curr; // advance the current state  
4     return *this;  
5 }
```

- ▶ Note that `this` is a special word that means the pointer to the object of this class
- ▶ `*this` dereferences the object, returning a reference to the `StrBlobPtr` object itself
- ▶ To allow access to the data member, our pointer class needs to be a `friend` of `StrBlob`
- ▶ Member functions of a friend class can access *all* members of the class that declares the other one a friend

Notes on StrBlobPtr

- ▶ We can also give `StrBlob` `begin` and `end` operations to allow for iterator use:

```
1 // forward declaration needed for
2 // friend declaration in StrBlob
3 class StrBlobPtr;
4 class StrBlob {
5     friend class StrBlobPtr;
6     // other members as before ...
7     // return StrBlobPtr to the first and one past
8     // the last elements
9     StrBlobPtr begin() { return StrBlobPtr(*this); }
10    StrBlobPtr end() {
11        auto ret = StrBlobPtr(*this, data->size());
12        return ret; }
13};
```

new and Arrays

- ▶ May need ability to allocate storage from many objects at once
- ▶ For example vectors and strings may need to allocate several elements at once
- ▶ Can do this with second kind of `new` expression than allocates and initializes objects
- ▶ Second method is to use `allocator` that separates allocation from initialization
- ▶ `allocator` generally provides better performance
- ▶ *Most applications should use a library container rather than dynamically allocated arrays*
- ▶ However, you will find much legacy code before 2011 that uses dynamically allocated arrays!
- ▶ Also, classes that use containers can use default copy, assignment and destruction
- ▶ Classes allocating dynamic arrays must define their own versions of these to manage memory

new and Arrays

- ▶ We tell `new` how many objects of type to allocate by giving number inside square brackets after the type name:

```
1 // call get_size to determine how many ints to
  // allocate
2 // pia points to the first of these ints
3 int *pia = new int [get_size()];
```

- ▶ Number inside square brackets need not be constant
- ▶ We can allocate array using type alias (and omit square brackets):

```
1 // arrT names the type array of 42 ints
2 typedef int arrT [42];
3 // allocates an array of 42 ints; p points to the
  // first
4 int *p = new arrT;
5 // above is equivalent to
6 int *p = new int [42];
```

Notes on `new` and Arrays

- ▶ Common to refer to memory allocated by `new T[]` as a dynamic array
- ▶ Actually it is just a pointer to the first element of the type of array
- ▶ Since it is not an array type, we can't use `begin` and `end` on a dynamic array
- ▶ For same reason, can't use range for loop on dynamic array
- ▶ Examples of initialization of dynamic arrays:

```
1 // block of ten uninitialized ints:
2 int *pia = new int[10];
3 // block of ten ints value initialized to 0:
4 int *pia2 = new int[10]();
5 // block of ten empty strings:
6 string *psa = new string[10];
7 // block of ten empty strings:
8 string *psa2 = new string[10]();
```

List initialization for new and Arrays

- ▶ In C++ 11 also allowed to provide braced list of elements in initialization:

```
1 // block of ten ints each initialized from
2 // the corresponding initializer
3 int *pia3 = new int [10]{0,1,2,3,4,5,6,7,8,9};
4 // block of ten strings; the first four are
5 // initialized from the given initializers
6 // remaining elements are value initialized
7 string *psa3 = new string [10]{ "a", "an", "the",
    string(3, 'x') };
```

- ▶ Can supply fewer initializer elements than array length – remaining elements are default initialized

More on `new` and Arrays

- ▶ As we saw, we can use expression to determine number of objects to allocate:

```
1 // get_size returns the number of elements needed
2 size_t n = get_size();
3 // allocate an array to hold the elements
4 int* p = new int[n];
5 for (int* q = p; q != p + n; ++q) {
6     /* process the array */ ;
7 }
```

- ▶ What happens when `get_size()` returns 0?
- ▶ Surprisingly code works fine, and it is legal!
- ▶ Best not to dereference `p` however, as it will be null
- ▶ Luckily for loop will not execute any iterations

Freeing dynamic arrays

- ▶ Also have a special form of `delete` that includes empty pair of square brackets to delete dynamic array:

```
1 // p must point to a dynamically allocated object  
  or be null  
2 delete p;  
3 // pa must point to a dynamically allocated array  
  or be null  
4 delete [] pa;
```

- ▶ Note that square brackets are required, but compiler is unlikely to warn us if we forget them
- ▶ Even for dynamic array allocated using type definition `arrT` must use square brackets

Smart pointers and dynamic arrays

- ▶ Library provides version of `unique_ptr` to manage arrays allocated by `new`
- ▶ Example use:

```
1 // up points to an array of ten
2 // uninitialized ints
3 unique_ptr<int []> up( new int [10] );
4 for (size_t i = 0; i != 10; ++i) {
5     // assign a new value to each of the elements
6     up[i] = i;
7 }
8 // automatically uses delete [] to destroy its
9   pointer
10 up.release();
```

- ▶ Note use subscript operator to access elements in array

Smart pointers and dynamic arrays

- ▶ If we want to use `shared_ptr`, need to provide our own deleter
- ▶ We can do that using a lambda to tell it to use `delete []`:

```
1 // to use a shared_ptr we must supply a deleter
2 shared_ptr<int> sp(new int [10],
3                   [] (int *p) { delete [] p; });
4 // uses the lambda we supplied that
5 // uses delete [] to free the array
6 sp.reset();
```

- ▶ Also no direct support, so accessing elements in array becomes:

```
1 // shared_ptrs don't have subscript operator
2 // and don't support pointer arithmetic
3 for (size_t i = 0; i != 10; ++i) {
4     // use get to get a built-in pointer
5     *(sp.get() + i) = i;
6 }
```

The allocator class

- ▶ Allows us to decouple memory allocation from construction
- ▶ Allows allocating large chunks of memory without overhead of constructing objects
- ▶ For example:

```
1 // construct n empty strings
2 string *const p = new string[n];
3 string s;
4 // q points to the first string
5 string *q = p;
6 while (cin >> s && q != p + n) {
7     *q++ = s; // assign a new value to *q
8 }
9 // remember how many strings we read
10 const size_t size = q - p;
11 // use the array
12 delete [] p;
```

The allocator class

- ▶ In previous example allocated and initialized `n` strings
- ▶ However, we input strings from `cin` so may not get `n` strings
- ▶ Also never used default value of strings – set value after `cin`
- ▶ Needed to write to each element twice – once to default init – then to assign value to them
- ▶ Also – classes without default constructor cannot be dynamically allocated as an array

The allocator class table of operations

<code>allocator<T> a</code>	define allocator to allocate objects of type T
<code>a.allocate(n)</code>	allocates n objects of type T
<code>a.deallocate(p,n)</code>	deallocate n objects of type T start at T* pointer p user must call destroy before deallocate
<code>a.construct(p,args)</code>	constructs object of type T at p using args
<code>a.destroy(p)</code>	runs destructor on the object pointed to by the T* pointer p

Use of allocator class

- ▶ The `allocator` is a template, so we must provide it type of object it can allocate in the angle brackets
- ▶ For example a string allocator:

```
1 // object that can allocate strings
2 allocator<string> alloc;
3 // allocate n unconstructed strings
4 auto const p = alloc.allocate(n);
```

- ▶ We use this allocated memory by constructing objects
- ▶ `construct` initializes object being constructed at specified location using zero or more additional elements
- ▶ additional elements must match a constructor for that class

Use of allocator class

► Example use of construct:

```
1 allocator<string> alloc ;
2 auto const p = alloc.allocate(n);
3 auto q = p; // get non-const pointer we can
  increment:
4 // below we construct empty string
5 alloc.construct( q++ ); // increment after
  construct
6 // below we construct string with 10 c's:
7 alloc.construct( q++, 10, 'c' );
8 // below construct string "hi":
9 alloc.construct( q++, "hi" );
10 // it is an error to use raw un-constructed memory:
11 cout << *p << endl; // ok: uses empty string
12 cout << *q << endl; //error: q points to
  unconstructed mem
```

Destroying elements using allocator class

- ▶ Example use of `destroy`:

```
1 // free the strings we actually allocated
2 while (q != p){ // decrement before destroy
3     alloc.destroy(--q);
4 }
```

- ▶ Can only destroy elements that were constructed
- ▶ Once elements are destroyed, can reuse memory to construct other strings, or free the memory using `deallocate`:

```
1 alloc.deallocate(p, n);
```

- ▶ Pointer `p` must not be null, and point to memory allocated by `allocate`
- ▶ Size passed must be same size used in call to `allocate`

Algorithms to copy and fill un-initialized memory

- ▶ Two algorithms (two versions of each) are in library to construct objects in uninit memory:

```
uninitialized_copy( b, e, b2 )
```

Copies elements in iterator range b, e into raw memory at b2. Memory at b2 must be large enough to hold elements

```
uninitialized_copy_n( b, n, b2)
```

Copy n elements starting at iterator b into memory at b2
Memory must be large enough to hold copy of elements

```
uninitialized_fill( b, e, t )
```

Construct objects in range of raw memory of iterators b, e as a copy of t

```
uninitialized_fill_n( b, n, t )
```

Construct n objects in raw memory starting at iterators b as a copy of t

A text query program

- ▶ As an example use of standard library, implement simple text-query program
- ▶ Count number of times a set of words occur and list of lines on which word occurs
- ▶ If word occurs more than once, only display line once
- ▶ Display lines in ascending order (ie. smaller line numbers before larger ones)
- ▶ For example searching a chapter of textbook for word “element” might yield:

```
element occurs 112 times
```

```
(line 36) A set element contains only a key;
```

```
(line 158) operator creates a new element
```

```
(line 160) Regardless of whether the element
```

```
(line 168) When we fetch an element from a map, we
```

```
(line 214) If the element is not found, find returns
```

```
... followed by ~100 more lines
```

A text query program design

- ▶ Read input lines one at a time – break into words – remember lines where word appears
- ▶ Generate output:
 - ▶ Fetch line numbers associated with given word
 - ▶ Make line numbers appear in ascending order without duplicates
 - ▶ Print text and line number
- ▶ Above can be met using following concepts from standard library – some new to you:
 - ▶ `vector<string>` to store copy of input file – one string per line – can get line number from index in vector
 - ▶ `istringstream` to break line into words
 - ▶ `set` to hold line numbers on which each word in the input appears – sorts line numbers and keeps them unique for each key in ascending order
 - ▶ `map` to associate each word with the set of line numbers – allows us to fetch `set` for any given word.

A text query data structures

- ▶ Design a class to hold input file instead of directly using `vector`, `set` and `map`
- ▶ We will make class `TextQuery` to hold `vector` with input file and `map` to associate each word in the file to the set of line numbers on which the word appears
- ▶ `TextQuery` constructor reads given input file
- ▶ `TextQuery` has method to perform queries
- ▶ Return query result as `QueryResult`, that has a print function
- ▶ `QueryResult` will “share” data, with `TextQuery`, so we will use `shared_ptrs`

Example use of proposed classes

```
1 void runQueries(istream &infile) {
2     // infile is an ifstream that is the file we want
3     // to query store the file and build the query map
4     TextQuery tq(infile);
5     // iterate with the user: prompt for a word
6     // to find and print results
7     while (true) {
8         cout << "enter word to look for, or q to quit: ";
9         string s;
10        // stop if we hit end-of-file on the input
11        // or if a 'q' is entered
12        if (!(cin >> s) || s == "q") break;
13        // run the query and print the results
14        print(cout, tq.query(s)) << endl;
15    }
16 }
```

TextQuery class design

- ▶ Data members of `TextQuery` class need to account for intended sharing of data with `QueryResult` class
- ▶ Will share two parts of the data:
 - ▶ the vector representing input file, and
 - ▶ sets that hold line numbers associated with each word
- ▶ The `map` associates each word in the file with dynamically allocated `set` holding line numbers
- ▶ define type member to refer to line numbers (indices into vector)

TextQuery class interface

- ▶ Note we have suppressed `std::` that should be in front of each of `std` library containers in header file interface

```
1 // declaration needed for return type
2 // in the query function
3 class QueryResult;
4 class TextQuery {
5 public:
6     using line_no = vector<string>::size_type;
7     TextQuery(std::ifstream&);
8     QueryResult query(const string&) const;
9 private:
10    // input file
11    shared_ptr< vector<string> > file;
12    // map of each word to the set of the
13    // lines in which that word appears
14    map< string , shared_ptr< set<line_no> > > wm;
15 };
```

TextQuery class constructor

```
1 // read input file; build map line numbers
2 TextQuery::TextQuery(istream &is) :
3     file( new vector<string> ) {
4     string text;
5     while ( getline(is, text) ) { // next line
6         file->push_back(text); // save line of text
7         // the current line number
8         int n = file->size() - 1;
9         // separate the line into words
10        istringstream line(text);
11        string word;
12        while (line >> word) { // next word in line
13            // if word not in wm, [word] adds empty entry
14            auto &lines = wm[word]; // lines is a shared_ptr
15            // that pointer is null the first time we see
16            word
17            if (!lines) // allocate a new set
18                lines.reset(new set<line_no>);
19            lines->insert(n); // insert this line number
20        }
21    }
```

Notes on TextQuery class constructor

- ▶ constructor allocates new `vector` to hold text from input file
- ▶ `getline` used to read line from file, and push it onto vector
- ▶ We use `->` operator to dereference `shared_ptr` of `file` to fetch the `push_back` member of vector
- ▶ `istringstream` in inner while loop is used to get next word in line we just read
- ▶ We use `map` subscript operator to fetch the `shared_ptr<set>` associated with `word`
- ▶ If `word` is not in `map`, subscript operator adds `word` to `map` `wm`, and element associated with that word is initialized as null pointer
- ▶ If `lines` is null, add a new `set` and call `reset` to update the `shared_ptr`

QueryResult class interface

- ▶ Note we have suppressed `std::` that should be in front of each of std library containers in header file interface

```
1 class QueryResult {
2     friend ostream& print( ostream&, const QueryResult&);
3 public:
4     QueryResult( string s, shared_ptr< set<line_no> > p,
5                 shared_ptr< vector<string> > f) :
6         sought(s), lines(p), file(f) { }
7 private:
8     string sought; // word this query represents
9     shared_ptr< set<line_no> > lines; // lines it's on
10    shared_ptr< vector<string> > file; // input file
11 };
```

Notes on TextQuery / QueryResult classes

- ▶ Note that we declare the `print` function to be a `friend` of `QueryResult`
- ▶ Functions or classes declared `friend` in a class are able to access the private methods and data of the class
- ▶ `QueryResult` Constructor just stores corresponding data members in initializer list
- ▶ `TextQuery::query()` function takes a string which it uses as the “key” to look in the `map` for the `set` of line numbers
- ▶ What happens if that string is not found?
- ▶ We will use a static `shared_ptr` to an empty `set` of line numbers and return a copy of that.

query method of TextQuery class

```
1 QueryResult
2 TextQuery::query(const string &sought) const
3 {
4     // we'll return a pointer to this set if we don't
5     // find sought
6     static shared_ptr< set<line_no> > nodata(new
7     set<line_no>);
8     // use find and not a subscript to avoid adding
9     // words to wm!
10    auto loc = wm.find(sought);
11    if (loc == wm.end()) {
12        return QueryResult(sought, nodata, file); // not
13        found
14    } else {
15        return QueryResult(sought, loc->second, file);
16    }
17 }
```

print QueryResult

```
1 ostream &print(ostream & os, const QueryResult &qr)
2 {
3     // if word was found, print count and all occurrences
4     os << qr.sought << " occurs "
5         << qr.lines->size() << " "
6         << make_plural(qr.lines->size(), "time", "s")
7         << endl;
8     // print each line in which the word appeared
9     for (auto num : *qr.lines) { // next element in set
10        // count text lines starting at 1
11        os << "\t(line " << num + 1 << ") "
12            << qr.file->at( num ) << endl;
13    }
14    return os;
15 }
```

Notes on print QueryResult

- ▶ Use the **size** of the **set** that we get from `qr.lines` to report how many times the word was found
- ▶ The **set** is in a `shared_ptr` so we need to use `->` to dereference the members of the object
- ▶ The function `make_plural` is called to print “time” or “times” depending on whether the size is 1 or more:

```
1 // return the plural version of word if ctr is
   greater than 1
2 string make_plural(size_t ctr, const string &word,
3                       const string
4                           &ending){
5     return (ctr > 1) ? word + ending : word;
6 }
```

- ▶ The numbers in the **set** are the line numbers, to which we add 1 to make human friendly counting from 1.
- ▶ We use the line number to fetch the num'th element in the vector of strings that `file` points to

Outline

Algorithms and maps

Dynamic memory

Final exam format and topics

Final exam format

- ▶ Exam time limit is 3 hours – worth 30% of final grade
- ▶ Closed book, non-programmable calculator allowed
- ▶ Three sections:
 - ▶ multiple choice (30 questions totalling 30 marks),
 - ▶ short answer less than five lines (10 questions totalling 30 marks),
 - ▶ long answer more than five lines (5 questions totalling 40 marks)
- ▶ Questions may be short snippets of code, or questions about concepts
- ▶ Study recommendations: - review the course slides, review the homework problems, practice writing code by writing small programs based on the topics in the course slides

Final exam topics

- ▶ Flow control statements
- ▶ Variables and basic types (const, reference, pointer, struct)
- ▶ Functions, return values and argument passing (by reference, by pointer)
- ▶ Function overloading and operator overloading
- ▶ Function defaults, and pointers to functions
- ▶ Strings, vectors, arrays, multi-dim arrays, iterators
- ▶ Namespaces and using declarations
- ▶ Expressions: arithmetic, logical and relational, type conversions, and precedence
- ▶ Classes, const and non-const member functions, and enumerations
- ▶ Iostream, fstream, and sstream class use; stream manipulators
- ▶ Graphics shape classes, GUI code
- ▶ Inheritance: base and derived classes, abstract types, mutability
- ▶ Object oriented programming: derivation, virtual functions, private and protected members
- ▶ Sequential and associative containers
- ▶ Generic algorithms
- ▶ Dynamic memory management

End of Scientific Computing

- ▶ Reminder: Final exam is Dec. 12, 2017 at 18:00 in 2L14
- ▶ Study hard, and Good luck in your future endeavours!
- ▶ Remember to have fun solving computing problems with your new skills!