- **Exceptions, RAII, and Smart Pointers - Lecture Outline**
- **The RAII concept**
  - **Useful idea: objects can do useful work just in their constructors and destructors**
    - *Take advantage of how compiler guarantees that constructor and destructor will be called for local variables when entering and leaving a scope*
    - *Example: save output stream numerical formatting state - examples/IO_demos - use in project!*
      - // a RAII-concept class to save and restore the numerical format settings
        // See handout on output stream formatting.

```
class Cout_format_saver {
public:
        Cout_format_saver() :
                old_flags(cout.flags()), old_precision(cout.precision())
                {}
        ~Cout_format_saver()
                {
                        cout.flags(old_flags);
                        cout.precision(old_precision);
                }
private:
        ios::fmtflags old_flags;
        int old_precision;
};


// usage: foo needs to change the precision, etc, but caller needs them
// to be whatever they were upon return. Create an object before
// changing the settings; will automatically restore upon return.

void foo(double x1, double x2)
{
        Cout_format_saver s;

        cout << fixed << setprecision(2) << x1 << endl;
        cout << setprecision(8) << x2 << endl;
}
```

    - *Often see a similar class in GUI class libraries for saving/restoring the state of the graphical system - e.g. the pen settings for drawing.*
  - **RAII - generalization of the concept**
    - *Resource Allocation is Initialization*
    - *if you need a resource, allocate in an object's constructor*
      - deallocate in the objects destructor
    - *compiler will make sure that constructors and destructors get called when they should be.*
    - *IMPORTANT in the presence of Exceptions!!!*
  - **Micro example**
    - *what happens in this code if a subfunction or other code throws an exception?*
      - void foo(int n)
        {
                char * p = new char[n];

```
        do_stuff(p);
        do_some_more_stuff();
        /* etc */
        // done
        delete[] p;
}
```

- *how to fix the problem:*
  - ugly solution

```
void foo(int n)
{
        char * p = new char[n];
        try {
                do_stuff(p);
                do_some_more_stuff();
                /* etc */
                // done
                delete[] p;
                }
        catch(...) {
                delete[] p;
                throw;
                }
}
```

  - what if there are multiple resources allocated - can get a complicated pattern of clean-up
    - different allocations at different points where the code might fail
  - This sort of thing will happen any time you use an ordinary built-in type of pointer to hold the results of new
- *Using RAII concept - allocate with a constructor, let the destructor do the deallocation*
  - illustration for our char buffer example:

```
class Char_buffer {
private
        char * p;
public:
        char_buffer(int n) : p(new char[n]} {}
        ~char_buffer()
                {delete[] p;}
        // overload pointer operators
        operator char* () const {return p;}
        char * operator-> () const {return p;}

void foo(int n)
{
        Char_buffer p(n);  // use p like a char*
        do_stuff(p);
        do_some_more_stuff();
        strcpy(p, s); // use like regular pointer
        cout << p << end;
        /* etc */
        // done
}
        // memory deallocated no matter how we leave
```

  - dumb example because std::string or std::vector will do the same thing for you, and more.

- **RAII rule:**
  - *In the presence of exceptions, using the RAII approach can ensure that your code always deallocates resources when it should.*
- **Ways to get RAII logic**
  - *wrap resource allocation logic in constructor/destructor functions*
    - see Stroustrup for e.g. C FILE * logic
    - common idea for e.g. network connections, locks
  - *For memory, Use std containers instead of raw memory arrays*
    - e.g. string or vector instead of new char[n], etc.
  - *other std::objects behave properly*
    - e.g. file streams - destructor will close the file, release the resource
  - *For memory, use a managed pointer, or "smart pointer" - generalization of the char_buffer example*
    - can be used like a pointer but manages the pointed-to object for you, and automatically deletes it when the last smart pointer is destroyed.

- **Smart Pointers**
  - **There are a variety of "Smart Pointers"**
    - *Basic idea is simple, but turned out to be hard to identify and solve all the potential problems, so took awhile.*
    - *But idea is a class of objects that behave and can be used like pointers, but when the last smart pointer to the object disappears, the pointed to object is automatically deleted.*
      - reference-counting logic
      - no dangling pointers, no double-deletion
      - copy/assign just adjusts the reference counts.
      - objects are automatically cleaned up when the smart pointers have gone out of scope or have been set to point at something else or nothing - get deleted when nobody is interested in them any more.
    - *give essentially garbage-collection facility to C++*
      - pros & cons of garbage collection ...
        - an OLD technology - highly developed in LISP, the original garbage-collected language
        - BTW, there are replacements for new operator that give garbage collection capability for C++
          - sketch of how it could work
        - automates some things, but lose control of execution time - significant because garbage collection can take a lot of time.
          - usually run independently of the program logic - whenever memory runs low, which could happen at any time
      - smart pointers in C++ result in relatively small amounts of execution time that happen under program control - uses RAII logic
  - **NOTE: since smart pointers are not built-in C++ language constructs, but just classes, can only be used correctly by programming by convention - you have to follow the rules yourself, and compiler will probably not help you if you break the rules!**
    - *A good smart pointer implementation is easy to use correctly, harder to break the rules, but not possible to make it foolproof.*
  - **Basic concepts of smart pointers:**
    - *a templated class, where the type is the type of the pointed to object, with an internal stored pointer of that type.*
      - template <class T>
        class Smart_ptr {
        private:
        T * ptr;
        };
    - *overloaded operators allow the smart pointer to be used syntactically like a regular built-in pointer type (see similar for iterators)*
      - T& operator* () {return *ptr;} // dereference
      - T* operator-> () const {return ptr;} // indirection, arrow operator
    - *Ways to get at the internal pointer when needed, but these are very scary because they make it easy to undermine the operation of the smart pointer - some implementations won't even supply these*
      - Have to try to stay with smart pointers all the time if possible!
        - If you use a raw pointer to interact with an object alongside smart pointers, no help with making sure the raw pointer will always be valid
      - operator T*() const {return ptr;} // conversion to pointer type
        - supports implicit conversions, which can be a nasty surprise - need to be careful!

        - T* get() const {return ptr;} // accessor - somewhat safer - have to deliberately call it
        - Can make less necessary with additional templates that provides a casting function
    - *Pointed-to object must be allocated with new, because it will get deallocated with delete, so can't be a stack object.*
        - difficult to enforce, so must program by convention - follow the rules
        - If possible, allocate and initialize the smart pointer in a single standalone statement using the constructor:
            - Smart_ptr<Thing> ptr(new Thing);
        - assignment from a raw pointer might be disallowed:
            - the_smart_ptr = thing_ptr; // won't compile
        - can use the "named constructor" idiom (later) to help enforce this.
    - *Have to have some way of keeping track of how many smart pointers are pointing to the object, and delete the object when the last one goes away.*
- **Ways to do smart pointers**
    - *strict ownership - std::unique_ptr*
        - only one smart pointer can "own" the object at a time
        - move construction and assignment transfer the ownership
        - copy construction and assignment disallowed to prevent ambiguity
        - when the owner is destroyed, it deletes the pointed-to object.
        - std::auto_ptr was failed effort at this - C++98 didn't have move semantics, so it was a mess. Now deprecated, don't use. unique_ptr is "auto_ptr done right"
    - *Reference-counted, intrusive - object must contain a reference count - usually provided by inheriting from a class that goes with the Smart_pointer template.*
        - Each object when created gets a zero reference count.
            - Each smart pointer coming into existing or being set to point to the object increments the reference count
            - Each smart pointer ceasing to exist, or being reset to not point to this object, decrements the refence count
            - When the reference count goes to zero, the Smart_ptr code automatically delete the object
        - Go through an implementation I used while waiting for a better one to be made standard.
            - see the implementation below
    - *Reference counted, non - intrusive - allocate a counter object with new when constructed, carry pointer to it, delete when destructed.*
        - usually don't have to modify the class - the reference count is a separate "manager" object
        - One-one relationship between manager object and managed object; all smart pointers to an object also point to the manager object.
        - C++11 shared_ptr
    - *Another non-intrusive reference-counting approach:*
        - We actually only need to tell the difference between 0 and more than 0 in the count
        - Instead of allocating a counter, the smart pointers to the same object are in a linked list
            - assigning a linked_ptr to another causes it to get spliced into the list
            - assigning a linked_ptr to something else causes it to get spliced out of the list
            - if it was the last in the list, delete the pointed-to object

- **Smart_Pointer - an example of an intrusive smart pointer**
  - **Works OK, I've used it a lot, but it is not technically complete, and has no help for the cycle problem, and can be relatively easily misused compared to shared_ptr**
  - **The code - it's a template header file**
    - #ifndef SMART_POINTER_H
      #define SMART_POINTER_H

      /* An intrusive reference-counting Smart_Pointer class template
      see R.B. Murray, C++ Strategy and Tactics. Addison-Wesley, 1993.
      Modified to resemble std::shared_ptr<> in important ways.

      Usage:

      1. Inherit classes from Reference_Counted_Object

          class My_class : public Reference_Counted_Object {
          // rest of declaration as usual
          };

      2. Always allocate objects with new as a Smart_Pointer constructor
      argument.

              Smart_Pointer<const My_class> ptr(new My_class);

      3. Use Smart_Pointers with the same syntax as built-in pointers;
      will convert to Smart_Pointers of another type; can be stored in
      Standard Library Containers. Using the casting functions to
      perform standard casts. Assignment from a raw pointer is not allowed
      to help prevent programming errors.

      4. When the last Smart_Pointer pointing to an object is destructed,
      or reset, the pointed-to object will be deleted automatically.

      5. Don'ts:
      Never explicitly delete the pointed-to object; reset the Smart_Pointer instead.
      Never attempt to point a Smart_Pointer to a stack object.
      Don't use the get() accessor unless absolutely necessary.

      6. Don't interfere with the reference counting. This very old and simple design
      unfortunately puts the reference counting functions into the public interface.
      Only the Smart_Pointers should be calling these functions. If your own code calls
      them, you are violating the design concept and can easily produce undefined behavior.
      Don't do it.

      The effects of breaking any of these rules is undefined.
      */

      /*
      Reference_Counted_Objects should only be allocated using new, never the stack.
      Smart_Pointers should be the only class that calls the increment and decrement functions.
      If the use count hits zero as a result of decrement, the object deletes itself.
      The reference count is declared mutable to allow increment/decrement_ref_count
      to be declared const, so that a Smart_Pointer can point to a const object.
      This is consistent with the conceptual constness of the object - merely referring
      to an object with a Smart_Pointer should not force it to be non-const.

```
*/
class Reference_Counted_Object {
public:
	Reference_Counted_Object () : ref_count(0)
		{}
	Reference_Counted_Object (const Reference_Counted_Object&) : ref_count(0)
		{}
	virtual ~Reference_Counted_Object()
		{}
	void increment_ref_count() const
		{++ref_count;}
	void decrement_ref_count() const
		// suicidal - destroys this object
		{if (--ref_count == 0) delete this;}
	// Available for testing and debugging purposes only; not normally used.
	long get_ref_count() const
		{return ref_count;}
private:
	mutable long ref_count;
};

/* Template for Smart_Pointer class
Overloads *, ->, =, ==, and < operators.
Simply increments and decrements the reference count when Smart_Pointers
are initialized, copied, assigned, and destructed.
*/
template <class T> class Smart_Pointer {
public:
	// Constructor with pointer argument - copy and increment_ref_count count
	// Explicit to disallow implicit construction from a raw pointer.
	explicit Smart_Pointer(T* arg = 0) : ptr(arg)
		{if (ptr) ptr->increment_ref_count();}
	// Copy constructor - copy and increment_ref_count
	Smart_Pointer(const Smart_Pointer<T>& other): ptr(other.ptr)
		{if (ptr) ptr->increment_ref_count();}
	// Templated constructor to support implicit conversions to other Smart_Pointer type
	template <class U> Smart_Pointer(const Smart_Pointer<U> other) : ptr(other.get())
		{if (ptr) ptr->increment_ref_count();}
	// Destructor - decrement ref count
	~Smart_Pointer()
		{if (ptr) ptr->decrement_ref_count();}
	// Assignment by copy-swap will decrement lhs, increment rhs
	const Smart_Pointer<T>& operator= (const Smart_Pointer<T>& rhs)
		{
			Smart_Pointer<T> temp(rhs);
			swap(temp);
			return *this;
		}
	// Reset this Smart_Pointer to no longer point to the object.
	// Swap with default-constructed Smart_Pointer will decrement the ref count,
	// and set the internal pointer to zero.
	void reset()
		{
			Smart_Pointer<T> temp;
			swap(temp);
		}
```

```
        // The following functions are const because they do not change
        // the state of this Smart_Pointer object.
        // Access the raw pointer - use this with caution! - avoid if possible
        T* get() const {return ptr;}
        // Overloaded operators
        // Dereference
        T& operator* () const {return *ptr;}
        T* operator-> () const {return ptr;}
        // The following operators make accessing the raw pointer less necessary.
        // Conversion to bool to allow test for pointer non-zero.
        operator bool() const {return ptr;}
        // Smart_Pointers are equal if internal pointers are equal.
        bool operator== (const Smart_Pointer<T>& rhs) const {return ptr == rhs.ptr;}
        // Smart_Pointers are < if internal pointers are <.
        bool operator< (const Smart_Pointer<T>& rhs) const {return ptr < rhs.ptr;}
        // Swap contents with another Smart_Pointer of the same type.
        void swap(Smart_Pointer<T>& other)
                {T* temp_ptr = ptr; ptr = other.ptr; other.ptr = temp_ptr;}
private:
        T* ptr;
};

// Casting functions - simulate casts with raw pointers
// Usage: If ptr is a Smart_Pointer<From_type>
// Smart_Pointer<To_type> = static_Smart_Pointer_cast<To_type>(ptr);
// Smart_Pointer<To_type> = dynamic_Smart_Pointer_cast<To_type>(ptr);
template <typename To_type, typename From_type>
Smart_Pointer<To_type> static_Smart_Pointer_cast(Smart_Pointer<From_type> ptr)
        {return Smart_Pointer<To_type>(static_cast<To_type *>(ptr.get()));}

template <typename To_type, typename From_type>
Smart_Pointer<To_type> dynamic_Smart_Pointer_cast(Smart_Pointer<From_type> ptr)
        {return Smart_Pointer<To_type>(dynamic_cast<To_type *>(ptr.get()));}

#endif
```

- **Summary of Smart_Pointer class:**
  - **Can default construct, and construct from a raw pointer, or another Smart_Pointer, even those pointing to a different type, if the conversion of raw pointers is legal (e.g. an upcast)**
    - template<class T>
      class Smart_Pointer {
      public:
      Smart_Pointer(T* arg = 0);
      Smart_Pointer(Smart_Pointer<T> const & r);
      template<class U>
      Smart_Pointer(Smart_Pointer <U> const & r);
  - **Can assign only from another Smart_Pointer of the same type**
    - can't assign from a raw pointer for safety and reliability reasons - makes it easier to follow the rules
    - Smart_Pointer<T>& operator= (Smart_Pointer<T> const & r);
  - **Can clear (point to nothing) with reset()**
    - void reset();
  - **Can dereference, get the raw pointer, and treat as bool**
    - T & operator*() const;
      T * operator->() const;
      T * get() const; // use with caution!
      operator bool() const; // true if pointing to existing object
  - **Can access the reference count from the object's Reference_Counted_Object members:**
    - long Reference_Counted_Object::get_ref_count() const;
  - **Can compare to another Smart_Pointer of the same type in terms of the underlying pointer**
    - *Anything that you can do with the underlying pointer, you can do with the Smart_Pointer, pretty much.*
    - *Smart_Pointer*
    - bool operator== (Smart_Pointer <T> const & a);
      bool operator!= (Smart_Pointer <T> const & a);
      bool operator< (Smart_Pointer <T> const & a);
  - **Can convert to another legal type or apply a dynamic cast**
    - Basically does the corresponding casts on the internal pointer, and returns a shared_ptr to the same object. - so you don't have to access the raw pointer.
    - template< typename To, typename From>
      Smart_Pointer <To> static_Smart_Pointer_cast(Smart_Pointer <From> const & r);

      template< typename To, typename From >
      Smart_Pointer <To> dynamic_Smart_Pointer_cast(Smart_Pointer <From> const & r);
    - usage:
      Smart_Pointer<To_type> p = dynamic_Smart_Pointer_cast<To_type>(from_type_ptr);

- **Demo of Smart_Pointer**
  - /*
    Demonstration of Smart_Pointer template.
    */
    #include <iostream>
    #include "Smart_Pointer.h"

    using namespace std;

    /* A Thing has a pointer to another Thing and inherits from the Reference_Counted_Object
    class defined in Smart_Pointer.h.

    The following code creates two Things that point to each other, and hands a
    Smart_Pointer to "this" to another function. Because the reference count is kept
    in "this" object, no special arrangements need to be made to create a Smart_Pointer
    to "this".
    */

    class Thing;
    void print_Thing(Smart_Pointer<Thing> ptr); // declare this function here, define later

    // Thing inherits from a class that provides the reference counting functionality
    class Thing : public Reference_Counted_Object {
    public:
        // give each Thing a unique number so we can easily tell them apart
        Thing() : i(++count) {}
        // verify the destruction
        ~Thing () {cout << "Thing " << i << " destruction" << endl;}
        int get_i() const {return i;}
        // make this Thing point to another one
        void set_ptr(Smart_Pointer<Thing> p)
            {ptr = p;}
        // display who this Thing is now pointing to, if anybody
        void print_pointing_to() const
        {
            if(ptr)
                cout << "Thing " << i << " is pointing to Thing " << ptr->get_i() << endl;
            else
                cout << "Thing " << i << " is pointing to nobody" << endl;
        }

        // demonstrates handing a Smart_Pointer to another function
        void print_from_this()
        {
            // get a Smart_Pointer that shares owenership with other Smart_Pointers
            Smart_Pointer<Thing> p(this);
            print_Thing(p);
            // shorter:
            //print_Thing(Smart_Pointer<Thing>(this));
            // even shorter:
            //print_Thing(this);
        }

        void reset_pointer() // call to reset the internal pointer
        {

```
                ptr.reset();
        }

private:
        Smart_Pointer<Thing> ptr;      // keeps other Thing in existence as long as this Thing exists
        int i;
        static int count;
};

int Thing::count = 0;

// a function that takes a Smart_Pointer and calls the print_pointing_to function
void print_Thing(Smart_Pointer<Thing> ptr)
{
        cout << "in print_Thing: ";
        ptr->print_pointing_to();
}

int main()
{
        Smart_Pointer<Thing> p1(new Thing);
        Smart_Pointer<Thing> p2(new Thing);

        // create a cycle with two set_ptr calls:
        p1->set_ptr(p2);
        p2->set_ptr(p1);

        // display what each object is pointing to:
        p1->print_pointing_to();
        p2->print_pointing_to();

        // invoke a function that passes a pointer to the object to another function.
        p1->print_from_this();
        p2->print_from_this();

        // break the cycle by calling the pointer reset function for one of the objects
        // to discard its pointer. Without this, the two Things will not get destroyed.
        p1->reset_pointer();
        p2.reset();   // this assignment will immediately destroy the second Thing.

        cout << "Exiting main function" << endl;
        // The first Thing will be destroyed when p1 goes out of scope
}


/* Output
Thing 1 is pointing to Thing 2
Thing 2 is pointing to Thing 1
in print_Thing: Thing 1 is pointing to Thing 2
in print_Thing: Thing 2 is pointing to Thing 1
Thing 2 destruction
Exiting main function
Thing 1 destruction
*/
```

- **The Cycle Problem:**
  - **If two smart-pointered objects point to each other with a smart pointer, you can get cycles: the objects won't get deleted.**
  - **TANSTAFL principle - there ain't no such thing as a free lunch.**
  - **Clunky way to fix:**
    - *When program knows that it is time for objects to go away, break the cycles by telling each object to reset or zero its smart pointers. It works, but clunky because you have to remember to write this code and be sure it gets executed - not very automatic!*
- **Weak pointers - a better fix of the Cycle Problem**
  - **Weak pointers observe the object, but don't keep it alive.**
    - *Associated with the smart pointers - don't affect the reference count, but observe the reference count, and so know when the object is gone. Use shared_ptr for shared ownership, weak_ptr for "observing" an object.*
    - *Must check a weak pointer for validity before using it, and then make sure the pointed-to object stays around while it is being used.*
    - *Implemented in C++11 shared_ptr classes:*
      - shared_ptr - represents shared ownership of an object, a reference-counted smart pointer.
        - involves a manager object holding the reference count.
      - weak_ptr - points to the manager object and so can observe whether the managed object still exists.
        - weak_ptr is expired if object has already been deleted.
      - To make reliable, have to arrange so that weak_ptrs always have a meaningful & defined status.
    - *In C++11, done as follows:*
      - Can only create a weak_ptr from a shared_ptr.
      - Can't use a weak_ptr directly, have to get a shared_ptr from it instead.
      - A special function, weak_ptr::lock() creates a shared_ptr from pointed-to object, resulting in another shared_ptr to the same object, "locking" it into existence if it still exists.
      - If object is gone, returned pointer will test 0/false, so you know not to refer to it.
      - The shared_ptr goes out of scope when you are done, "unlocking" the object, allowing it to disappear if nobody else is pointed to it with a shared_ptr pointer.
      - weak_ptr has expired() member function that returns true if the pointed-to object no longer exists
      - Can also construct a shared_ptr from a weak_ptr, but an exception is thrown if the weak_ptr has expired.
  - **A concept for how to implement shared_ptr and weak_ptr**
    - *Non-intrusive, using a dynamically allocated reference counting "manager" object*
    - *the reference count object holds both a shared_ptr reference count, a weak_ptr reference count, and a pointer to the object.*
    - *creating/copying a shared_ptr to the object increments that ref count, destroying a shared_ptr decrements it.*
    - *creating/copying a weak_ptr to the object does the same thing with the weak_ptr reference count*
    - *if the shared_ptr reference count goes to zero, the pointed-to object is deleted.*
    - *if the weak_ptr reference count goes to zero, and the shared_ptr reference count is also already zero, then the manager object can be deleted.*
    - *the weak_ptr::lock() function checks the shared_count to determine whether the object still exists.*
  - **Note:**

- *Intrusive weak pointers are hard to implement because you want the object to disappear when the life-time affecting pointers are all gone, so you no longer have any place to look to see that it is.*
- *The link-list idea for reference counting also is hard to make work for weak pointers.*

- **Summary of C++11 shared_ptr class:**
  - **See tutorial handout posted on course web site - check it before looking elsewhere!**
  - **Careful design to help reduce errors while making it as easy to use as possible.**
  - **Can default construct, and construct from a raw pointer, a weak_ptr, or another shared_ptr, even those pointing to a different type, if the conversion of raw pointers is legal (e.g. an upcast)**
    - template<class T>
      class shared_ptr {
      public:
      shared_ptr();

      template<class Y>
      explicit shared_ptr(Y * p);

      shared_ptr(shared_ptr const & r);

      template<class Y>
      shared_ptr(shared_ptr<Y> const & r);

      template<class Y>
      explicit shared_ptr(weak_ptr<Y> const & r);
  - **Can assign only from another shared_ptr, with legal implicit conversions also**
    - can't assign from a raw pointer for safety and reliability reasons - makes it easier to follow the rules
    - shared_ptr & operator= (shared_ptr const & r);

      template<class Y>
      shared_ptr & operator= shared_ptr<Y> const & r);
  - **Can clear (point to nothing) with reset(), or reset to point to another raw pointed-object, as long as pointer type is convertible**
    - void reset();
    - template<class Y>
      void reset(Y * p);
  - **Can dereference, get the raw pointer, and treat as bool**
    - T & operator*() const;
      T * operator->() const;
      T * get() const;

      template<class T>
      T * get_pointer(shared_ptr<T> const & p);

      operator unspecified_bool_type const; // true if pointing to existing object
    - unspecified_bool_type is a trick to give you something that will test true/false but can't be accidentally implicitly converted to something else like a simple conversion to bool would.
  - **Can access the reference count**
    - bool unique() const;
      long use_count() const;
  - **Can compare in terms of the underlying pointer and its legal conversions**
    - *Anything that you can do with the underlying pointer, you can do with the shared_ptr, pretty much.*
    - template<class T, class U>
      bool operator== (shared_ptr<T> const & a, shared_ptr<U> const & b);

      template<class T, class U>

```
bool operator!= (shared_ptr<T> const & a, shared_ptr<U> const & b);

template<class T, class U>
bool operator< (shared_ptr<T> const & a, shared_ptr<U> const & b);
```

- **Can convert to another legal type or apply a dynamic cast**
  - Basically does the corresponding casts on the internal pointer, and returns a shared_ptr to the same object. - so you don't have to access the raw pointer.
  - template<class T, class U>
    shared_ptr<T> static_pointer_cast(shared_ptr<U> const & r);

    template<class T, class U>
    shared_ptr<T> const_pointer_cast(shared_ptr<U> const & r);

    template<class T, class U>
    shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const & r);

- **Summary of C++11 weak_ptr class:**
  - **Can default construct and construct from a another weak_ptr, or a shared_ptr, even those pointing to a different type, if the conversion of raw pointers is legal (e.g. an upcast)**
    - template<class T> class weak_ptr {
      public:
      weak_ptr();

      weak_ptr(weak_ptr const & r);

      template<class Y>
      weak_ptr(shared_ptr<Y> const & r);

      template<class Y>
      weak_ptr(weak_ptr<Y> const & r);
  - **Can assign only from another weak_ptr or a shared_ptr, even of different legal type**
    - weak_ptr & operator=(weak_ptr const & r);

      template<class Y>
      weak_ptr & operator= (weak_ptr<Y> const & r);

      template<class Y>
      weak_ptr & operator= (shared_ptr<Y> const & r);
  - **Can clear (point to nothing) with reset()**
    - void reset();
  - **Can check whether the pointed-to object is gone, or create a shared_ptr to lock the pointed-to object into existence if it still exists**
    - bool expired() const; // true if the pointed-to object no longer exists

      shared_ptr<T> lock() const; // return a shared_ptr; will be zero if object does not exist

- **Special problem: Getting a shared_ptr to "this" object**
  - **Say you have objects being managed by a shared_ptr - Thing objects.**
  - **Say you want to call a function and give it a shared_ptr to "this" object:**
    - void Thing::call_another()
      {
      foo(shared_ptr<Thing>(this));
      }
  - **Problem:**
    - The shared_ptr constructor taking the raw this pointer here will start a new reference counting manager object for Thing, meaning that there are now two shared_ptr manager objects trying to control the same object - a double deletion will result - crash!
      - Don't have this problem with an intrusive pointer like Smart_Pointer because the reference count is in "this" object, so starting a new Smart_Pointer from the raw this pointer is no problem.
  - **Solution: Set up a weak_ptr member variable pointing to this object - INTRUSIVE**
    - Arrange so that when a Thing object is constructed, it contains a weak_ptr member variable. Then when the first regular shared_ptr is created, the shared_ptr template by magic notices the existence of the weak_ptr and initializes it from itself. Then if you need to give some other object a shared_ptr, it can also be initialized from the weak_ptr member variable. This ensures that only one manager object is ever involved (provided you follow the rules).
  - **Making it easy:**
    - Clumsy to do stand-alone, but C+11 includes a template class, enable_shared_from_this<T>, that you can inherit from which sets up a weak_ptr member variable. The shared_ptr constructor checks for it and will initialize the weak pointer. Then a member function from the template class, shared_ptr<T> shared_from this() will return a shared_ptr based on the weak pointer.
      - Example:
        - class Thing : public enable_shared_from_this<Thing> {

          void call_other_function()
          {
          shared_ptr<Thing> p = shared_from_this();
          other_function(p);
          }
          };
      - see more example code below
    - Negative: To use weak_ptr with shared_from_this(), you have to modify the class by inheriting from enable_shared_from_this<T> - instrusive!
    - A Gotcha
      - *Oops!* The weak_ptr member variable that your class gets from enable_shared_from_this<> does not get initialized until the first shared_ptr pointing to the object is constructed, which can only happen after your class's constructor finishes executing. This means that you can't use shared_from_this() in your class's constructor! If you do, you will get a bad weak_ptr exception.
  - **Demo of C++11 shared_ptr, also shows cycle problem**
    - /*
      Demonstration of shared_ptr.
      This demo shows a how shared_ptr works in various situations with and without a cycle problem.

      */

      #include <iostream>
      #include <memory>

```
using namespace std;

/* A Thing has a pointer to another Thing. The following code creates two Things that
point to each other. If the internal pointers are shared_ptrs, the resulting
cycle means that the two objects will not get deleted even though main has discarded
their pointers.
*/

class Thing {
public:
	// give each Thing a unique number so we can easily tell them apart
	Thing() : i(++count) {}
	// verify the destruction
	~Thing () {cout << "Thing " << i << " destruction" << endl;}
	int get_i() const {return i;}
	// make this Thing point to another one
	void set_ptr(shared_ptr<Thing> p)
		{ptr = p;}
	// display who this Thing is now pointing to, if anybody
	void print_pointing_to() const
	{
		if(ptr)
			cout <<  "Thing " << i << " is pointing to Thing " << ptr->get_i() << endl;
		else
			cout <<  "Thing " << i << " is pointing to nobody" << endl;
	}
private:
	shared_ptr<Thing> ptr;  // keeps other Thing in existence as long as this Thing exists
	int i;
	static int count;
};

int Thing::count = 0;

int main()
{
	/* Always create objects like this, in single stand-alone statement that does nothing
	more than create the new object in a shared_ptr constructor.
	*/

	shared_ptr<Thing> p1(new Thing);
	shared_ptr<Thing> p2(new Thing);

/*	// It is difficult to set a shared_ptr to an object any other way.
	// The definition of shared_ptr disallows assigning from a raw pointer!
	shared_ptr<Thing> p3;

	Thing * raw_ptr = new Thing;
	p3 = raw_ptr;		// disallowed! compile error!

	// the following works, but there is no protection against raw_ptr being used elsewhere; try to
avoid
	p3.reset(raw_ptr);
*/
```

```
        // create the cycle with two set_ptr calls:
        p1->set_ptr(p2);
//      p2->set_ptr(p1);

        // display what each object is pointing to:
        p1->print_pointing_to();
        p2->print_pointing_to();

//      reset with no arguments is how you discard the pointed-to object;
//      it zeroes-out the internal pointer to the shared object;
//      If there were no cycles, this will cause both Things to be deleted.
//      p1.reset();
        p2.reset();


        cout << "Exiting main function" << endl;
}


/* Sample output with both cycle-creation statements and reset statements commented out,
showing automatic destruction on return:
--------
Thing 1 is pointing to nobody
Thing 2 is pointing to nobody
Exiting main function
Thing 2 destruction
Thing 1 destruction
*/

/* Sample output with both cycle-creation statements commented out, and reset statements in,
showing destruction when both pointers are reset.
--------
Thing 1 is pointing to nobody
Thing 2 is pointing to nobody
Thing 1 destruction
Thing 2 destruction
Exiting main function
*/

/* Sample output with the only the first of the two cycle-creation statements in and executed,
and only the second pointer (p2) reset statement in and executed.
There is only a "half cycle" because Thing 1 is pointing to Thing 2, but Thing 2 points
to nobody. Discarding our pointer to Thing 2 does not destroy Thing 2 because it is kept alive
by Thing 1 pointing to it. When p1 goes out of scope Thing 1 is destroyed, which then results
in Thing 2 being destroyed. This shows smart pointers doing their automatical destruction.
--------
Thing 1 is pointing to Thing 2
Thing 2 is pointing to nobody
Exiting main function
Thing 1 destruction
Thing 2 destruction
*/

/* Sample output with cycle-creation statements in place and executed,
and reset statements either in or commented out (doesn't change the output).
Notice how neither object gets destroyed! They keep each other alive!
```

```
--------
Thing 1 is pointing to Thing 2
Thing 2 is pointing to Thing 1
Exiting main function
*/
```

- **Demo of C++11 weak_ptr, showing solution to cycle problem**
  - ```
    /*
    Demonstration of shared_ptr and weak_ptr, showing how cycle problems can be solved.
    */
    #include <iostream>
    #include <memory>
    using namespace std;

    /* A Thing has a pointer to another Thing. The following code creates two Things that
    point to each other. If the internal pointers are weak_ptrs, then when main discards
    its shared_ptrs, the objects get deleted properly.
    */

    class Thing {
    public:
        // give each Thing a unique number so we can easily tell them apart
        Thing() : i(++count) {}
        // verify the destruction
        ~Thing () {cout << "Thing " << i << " destruction" << endl;}
        int get_i() const {return i;}
        // make this Thing point to another one
        void set_ptr(shared_ptr<Thing> p)
            {ptr = p;}
        // display who this Thing is now pointing to, if anybody
        void print_pointing_to() const
        {
            if(ptr.expired())      // see if the pointed-to object is there
                cout << "Thing " << i << " is pointing at nothing" << endl;
            else {
                // create a temporary shared pointer, making sure the other Thing stays around
                // long enough to look at it
                shared_ptr<Thing> p = ptr.lock();
                if(p)    // redundant in this code, but another way to test for expiration
                    cout <<  "Thing " << i << " is pointing to Thing " << p->get_i() << endl;
                }
        }
    private:
        weak_ptr<Thing> ptr;     // points to the other Thing, but doesn't affect lifetime
        int i;
        static int count;
    };

    int Thing::count = 0;

    int main()
    {
        /* Always create objects like this, in single stand-alone statement that does nothing
        more than create the new object in a shared_ptr constructor, to minimize any possibility
        of having a stray ordinary pointer involved, or other wierd effects.
        */
    ```
    ```

```
        shared_ptr<Thing> p1(new Thing);
        shared_ptr<Thing> p2(new Thing);

        // display what each object is pointing to:
        p1->print_pointing_to();
        p2->print_pointing_to();

        // create the cycle with two set_ptr calls:
        p1->set_ptr(p2);
        p2->set_ptr(p1);

        // display what each object is pointing to:
        p1->print_pointing_to();
        p2->print_pointing_to();

/*      A.
        // reset with no arguments is how you discard the pointed-to object;
        // it zeroes-out the internal pointer to the shared object;
        p1.reset();
        // display what Thing 2 is pointing to
        p2->print_pointing_to();
*/

        cout << "Exiting main function" << endl;
// when p1 and p2 go out of scope, they will free the pointed-to objects if they are the
// last pointers referring to them.
}

/* Sample output with the cycle created and the reset/print statements in A commented out.
Both objects are deleted on exit in spite of the cycle.
--------
Thing 1 is pointing at nothing
Thing 2 is pointing at nothing
Thing 1 is pointing to Thing 2
Thing 2 is pointing to Thing 1
Exiting main function
Thing 2 destruction
Thing 1 destruction
*/

/* Sample output with the cycle created and the statements in A executed.
Thing 1 gets destructed by the reset, and now Thing 2 knows that it is not
pointing at anything any more. Exiting then destroys Thing 2 also.
--------
Thing 1 is pointing at nothing
Thing 2 is pointing at nothing
Thing 1 is pointing to Thing 2
Thing 2 is pointing to Thing 1
Thing 1 destruction
Thing 2 is pointing at nothing
Exiting main function
Thing 2 destruction
*/
```

- **Demo of C++11 shared_from_this()**
  - **Based on previous examples**
  - /*
    Demonstration of shared_ptr's shared_from_this facility.
    */

    #include <iostream>
    #include <memory>

    using namespace std;

    /* A Thing has a pointer to another Thing. The following code creates two Things that
    point to each other.
    */

    // shared_ptr works with an incomplete type!
    class Thing;
    void print_Thing(shared_ptr<Thing> ptr); // declare this function here, define later

    // Thing inherits from template class enable_shared_from_this<>
    class Thing : public enable_shared_from_this<Thing> {
    public:
        // give each Thing a unique number so we can easily tell them apart
        Thing() : i(++count) {}
        // verify the destruction
        ~Thing () {cout << "Thing " << i << " destruction" << endl;}
        int get_i() const {return i;}
        // make this Thing point to another one
        void set_ptr(shared_ptr<Thing> p)
            {ptr = p;}
        // display who this Thing is now pointing to, if anybody
        void print_pointing_to() const
        {
            if(ptr)
                cout << "Thing " << i << " is pointing to Thing " << ptr->get_i() << endl;
            else
                cout << "Thing " << i << " is pointing to nobody" << endl;
        }

        // demonstrates use of shared_from_this()
        void print_from_this()
        {
            // get a shared_ptr that shares owenership with other shared_ptrs
            shared_ptr<Thing> p = shared_from_this();
            print_Thing(p);
            // shorter:
            //print_Thing(shared_from_this());
        }

        void reset_pointer()   // call to reset the internal pointer
        {
            ptr.reset();
        }

```
private:
	shared_ptr<Thing> ptr;  // keeps other Thing in existence as long as this Thing exists
	int i;
	static int count;
};

int Thing::count = 0;

// a function that takes a shared_ptr and calls the print_pointing_to function
void print_Thing(shared_ptr<Thing> ptr)
{
	cout << "in print_Thing: ";
	ptr->print_pointing_to();
}

int main()
{
	/* Always create objects like this, in single stand-alone statement that does nothing
	more than create the new object in a shared_ptr constructor. This will help prevent
	accidentally assigning more than one shared_ptr family to the same object.
	*/

	shared_ptr<Thing> p1(new Thing);
	shared_ptr<Thing> p2(new Thing);

	// create a cycle with two set_ptr calls:
	p1->set_ptr(p2);
	p2->set_ptr(p1);

	// display what each object is pointing to:
	p1->print_pointing_to();
	p2->print_pointing_to();

	// invoke a function that uses shared_from_this to pass a pointer to the object
	// to another function.
	p1->print_from_this();
	p2->print_from_this();

	// reset with no arguments is how you discard the pointed-to object;
	// break the cycle by calling the pointer reset function for one of the objects
	// to discard its pointer
	p1->reset_pointer();
	p2.reset();

	cout << "Exiting main function" << endl;
}


/* Output
Thing 1 is pointing to Thing 2
Thing 2 is pointing to Thing 1
in print_Thing: Thing 1 is pointing to Thing 2
in print_Thing: Thing 2 is pointing to Thing 1
Thing 2 destruction
Exiting main function
Thing 1 destruction
```

```
*/
```

- **C++11's unique_ptr - "auto_ptr done right"**
  - **Has many features of shared_ptr, but …**
  - **Unique_ptr implements unique ownership of the pointed-to object - hence the name.**
    - *If used properly, only one unique_ptr can be pointing to an object at a time.*
    - *When the single "owning" unique_ptr is destroyed, the pointed to object gets deleted.*
    - *You can transfer the object ownership from one unique_ptr to another unique_ptr, but the first unique_ptr no longer points to the object and so no longer owns it.*
  - **Implementation is basically simple:**
    - *Has an internal pointer member variable and very little else.*
    - *If the internal pointer is zero (nullptr), it means that this unique_ptr doesn't own anything.*
    - *If non-zero, it means this unique_ptr owns the object being pointed to.*
    - *~unique_ptr() simply deletes using the internal pointer. If an object is owned, it gets deleted; otherwise nothing happens.*
    - *You can test the unique_ptr for true to see if it owns something - this just tests the internal pointer for non-zero.*
    - *Like other smart pointers, has dereference and -> operators to allow you to use it syntactically like a built-in pointer.*
    - *Like a built-in pointer, you can't validly dereference it if it is zero, so test first if it is possible that it no longer owns an object.*
    - *Like other C++11 smart pointers, you can initialize it from a raw (built-in) pointer only in the constructor, so follow the idiom putting the new expression in the constructor argument.*
      - unique_ptr<Thing> p(new Thing);
  - **Simple Use example:**
    - *Use to hold the pointer to a locally allocated object; the object will get deleted regardless of how we leave the function - much better than trying to ensure deletion using a raw pointer.*
      - void foo()
        {
        unique_ptr<Thing> p (new Thing);
        . . .
        p->defrangulate(); // use like an ordinary pointer

        . . . do stuff that might throw an exception

        . . . do stuff that might return from multiple places

        doesn't matter how we leave - since ~unique_ptr<Thing> will be executed for p, the new Thing will get deleted!
        }

      - If you want to delete the object earlier, you can call unique_ptr::reset(); this will delete the object and reset the internal pointer to nullptr - now looks just like a default constructed unique_ptr.
  - **What is special about unique_ptr compared to the other smart pointers?**
    - *The unique ownership concept is enforced by forbiding copy construction and copy assignment. You can't copy or assign unique_ptrs: For example:*
      - {
        unique_ptr<Thing> p1 (new Thing); // p1 owns a Thing
        unique_ptr<Thing> p2; // default ctor'd - p2 owns nothing

        p2 = p1; // compile error! can't assign!

```
unique_ptr<Thing> p3(p1);          // compile error! can't copy construct!
```

- *Because you can't copy a unique pointer, you will never get a situation in which two unique_ptrs think they own or point to the same object - no double deletion or dangling pointer is possible.*

- **Can I use a unique_ptr as a function argument?**
  - *Only if you pass it by reference - can't copy it!*
- **Can I use a unique_ptr as a function return value?**
  - *Yes! And it works very well for this - ownership is transferred out from the function to the receiving unique_ptr. Example:*
    - // create a Thing and return a unique_ptr to it

```
unique_ptr<Thing> create_Thing()
{
        unique_ptr<Thing> p (new Thing);
        return p;
}

void foo()
{
        unique_ptr<Thing> p1 (create_Thing);    // p1 now owns it.

        unique_ptr<Thing> p2;          // default construct
        p2 = create_Thing();           // assign from returned value

. . .
}
```

  - *Why does p2 = create_Thing(); work but the previous p2 = p1 wouldn't comple?*
    - That's what's special: in p2 = p1, the rhs is an lvalue (a variable with a name). In p2 = create_Thing(), the rhs is an rvalue (an unnamed temporary variable).
    - Ownership can be transfered implicitly from an rvalue unique_ptr, but not an lvalue unique_ptr.
- **How do you transfer ownership of an object between unique_ptrs?**
  - *It is done using move sematics. If a unique_ptr is an rvalue, it's data (the pointer value) can be moved to another unique_ptr. The first unique_ptr is set to nullptr (it no longer owns the object), and the second unique_ptr now owns the object.*
  - *So unique_ptr has move constructors and move assignment - the original or rhs is an rvalue-reference (&&).*
  - *Using a returned unique_ptr as above is simply a result of move operations being used for the returned value.*
  - *You can transfer ownership between named unique_ptrs by changing an lvalue unique_ptr into an rvalue unique_ptr using std::move. (Recall that std::move does a cast to rvalue-reference). Example:*
    - {

```
        unique_ptr<Thing> p1 (new Thing);  // p1 owns a Thing
        unique_ptr<Thing> p2;              // default ctor'd - p2 owns nothing

    // get an rvalue for p1, then can move assign - ownership transfers
        p2 = std::move(p1);  // p2 now owns the Thing, p1 now owns nothing

    // get an rvalue for p2, then can move construct - ownership transfers
        unique_ptr<Thing> p3(std::move(p2)); // p3 now owns it and p2 now owns nothing
```

  - *So you can transfer a pointer value between unique_ptrs, and the ownership goes with it. You just have to be explicit about it if the source is an lvalue; it happens automatically if the source is an rvalue.*
- **Can I put unique_pointers into a Standard container?**

- *Yes - an important advantage over autopointer.*
- *Just have to remember that ownership is now in the container (in the unique_ptr object in the container) and will stay there unless you move it out.*
- *Things to remember:*
  - You must fill the container with rvalue unique_ptrs, so the ownership goes into the unique_ptr in the container. Either declare an unnamed unique_ptr as the fill-function argument, or use std::move with a named unique_ptr.
  - If you erase an item in the container, you are destroying the unique_ptr, which will then delete the object it is pointing to.
  - If you empty the container, all of the pointed-to objects will be deleted when the unique_ptrs are destroyed.
  - If you refer to a unique_ptr in the container, you can use it to refer to the object (you may have to test it for nullptr first - see below).
  - As long as you leave the unique_ptr in the container, it retains ownership.
  - If you move the ownership out of the container item, the empty unique_ptr stays in the container. If you leave it there, your code may need to check for non-empty unique_ptrs before dererencing them.
- *Example: filling a Standard vector container with unique_ptrs;*
  - {

```
        vector<unique_ptr<Thing>> v;
        for(int i = 0; i < 4; i++) {
                unique_ptr<Thing> p(new Thing);
                v.push_back(std::move(p)); // make p into rvalue and move into v
                // alternate:
//              v.push_back(unique_ptr<Thing>(new Thing));
                }
        v[2]->defrangulate();    // use the unique_ptr in the container

//      v[2].reset();  // tell that unique_ptr to delete its object; v[2] is now empty unique_ptr
unique_ptr<Thing> p2 = std::move(v[2]); // p2 now owns it, v[2] is now empty unique_ptr

//      v[2]->defrangulate(); // run time error - v[2] no longer owns anything; has nullptr;

// tell all currently existing Things to defrangulate
        for(int i = 0; i < 4; i++) {
                if(v[i])  // make sure that unique_ptr owns an object
                        v[i]->defrangulate();
                else
                        cout << "Thing " << i << " no longer exists" << endl;
                }
}
// when v goes out of scope, it and its contents are destroyed and remaining pointed-to objects
are automatically deleted.
```

- *Example: filling a Standard map with <string, unique_ptr>*
  - {

```
        map<string, unique_ptr<Thing>> m;
// different ways to fill the container
        m["Adam"] = unique_ptr<Thing>(new Thing);
        m["Bob"] = unique_ptr<Thing>(new Thing);
```

```
        m["Carl"] = create_Thing();
        m.insert(make_pair("Dave", unique_ptr<Thing>(new Thing)));
        unique_ptr<Thing> p_load (new Thing);
        m.insert(make_pair("Edward", std::move(p_load)));

        for(auto it = m.begin(); it != m.end(); ++it) {
            if(it->second)
                cout << it->first << " has pointer to Thing " << it->second->get_i() << endl;
            else
                cout << it->first << " has pointer to nothing" << endl;
            }

        // get rid of one - the Thing gets deleted
        m.erase("Carl");  // the pair is erased, meaning unique_ptr is destroyed
        // empty the pointer at Dave
        m.find("Dave")->second.reset();

        for(auto it = m.begin(); it != m.end(); ++it) {
            if(it->second)
                cout << it->first << " has pointer to Thing " << it->second->get_i() << endl;
            else
                cout << it->first << " has pointer to nothing" << endl;
            }

        // move one out:
        unique_ptr<Thing> p = move(m["Bob"]);
        for(auto it = m.begin(); it != m.end(); ++it) {
            if(it->second)
                cout << it->first << " has pointer to Thing " << it->second->get_i() << endl;
            else
                cout << it->first << " has pointer to nothing" << endl;
            }
    } // all the remaining Things get deleted
```

- **When is a good vs bad idea to use smart pointers?**
  - **"No naked new" fad - instead of raw pointers, always use make_shared ...**
    - *but this imposes overhead or complexity - not always a good idea - see reservations in Stroustrup*
  - **Use DIY memory management if performance is critical, pointers passed as arguments or stored in containers a lot, and memory management is simple.**
    - *E.g. objects are always deleted in exactly one place, and all objects are quaranteed to be deleted.*
    - *No reference counting overhead, no unique-ownership to manage; built-in pointers fast to copy as function arguments*
  - **Use shared_ptr when:**
    - *When you actually, really have a situation with shared ownership and the DIY ownership managment is hard to get right.*
    - *You have cycles that would be complicated to solve without weak_ptr*
  - **Use unique_ptr**
    - *to automate deletion in simple cases - e.g where an object owns another object via unique_ptr member variable not made available outside the class.*
    - *when you want to represent unique ownership with several possible owners.*