- **Exceptions, RAII, and Smart Pointers - Lecture Outline**
- **More on exceptions and exception handling**
  - **Exceptions can be in class hierarchies, can catch with the base:**
    - *Tip: always catch an exception object by reference ...*
    - *not a bad idea: inherit from std::exception, override virtual char * what() const. Gives a uniform error reporting facility for all exceptions:*
      - try {
        /* stuff */
        }
        catch (exception& x)
        {
        cout << x.what() << endl;
        }
        catch (...)
        {
        cout << "unknown exception caught" << endl;
        }
    - *derived classes can build an internal string having whatever info in it that you want, return the .c_str() for what()*
  - **Special situations with exceptions**
    - *What happens if something goes wrong with constructing an object?*
      - e.g. if getting initialization data from a file, what if some of the data is invalid?
      - note that constructors have no return value, so no obvious way to signal that it didn't work.
      - without exceptions - only way to handle is to quit trying to construct the object and set some kind of member variable to say the object is no good, and then insist that client code check it before using the object.
        - object is actually a zombie - not fully initialized, but walks anyway? What do you do with it?
      - better approach is to throw an exception - forces an exit from the constructor function
    - *What happens if an exception is thrown during construction of an object?*
      - Any members that were successfully constructed are destroyed - their destructors are run
      - Any memory for the whole object that was allocated is deallocated.
      - Control leaves the constructor function at the point of the exception
    - *if an exception is thrown from a constructor, object does not exist!*
      - Example code:

        Thing * p;
        try { // try block defines a scope
        Thing t;
        p = new Thing;
        }
        catch(Thing_ctor_failure& x)
        {
        // what is status of Thing t or the p's pointed to Thing at this point and later?
        }

      - What is the status of Thing t and p's pointed-to Thing in the catch block and afterwards?
        - Thing t is out of scope now - can't refer to it anyway!

        - Guru Sutter describes this in terms of the Monty Python dead parrot sketch.
        - There never was a parrot - it was never alive!
        - Both t, and p's pointed-to thing never existed!
        - p does not point to a valid object, or even a usable object - actually no object at all - don't try to use it in any way, shape, or form
- *What about throwing an exception in a destructor?*
    - if an exception gets thrown out of a dtor, and we are already unwinding the stack, what is system supposed to do with the TWO exceptions now going on? rule: shouldn't happen!
        - if happens, terminate - exception handling is officially broken!
    - if exceptions might get thrown during destruction, you must catch them and deal with them yourself inside the dtor function before exiting it

- **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*
      - // 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 lat smart pointer is destroyed.

- **Smart Pointers**
  - **Can use 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 refernce counts, and follows normal semantics!
      - objects are automatically cleaned up when nobody is interested in them.
    - *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 - 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 ptr(new Thing);
    - assignment from a raw pointer might be disallowed:
      - 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::auto_ptr<>*
    - only one smart pointer can "own" the object at a time
    - when the owner is destroyed, it deletes the pointed-to object.
    - non-owners are left dangling
  - *Reference-counted, intrusive - must inherit from a class providing a reference count*
    - reference counting - a common idea
      - 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've been using while waiting for a better one to be made standard.
      - see the implementation below
  - *Reference counted, non - intrusive - allocate a counter 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
    - boost/tr1 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

- **std::auto_ptr - a very dumb and cranky smart pointer**
  - **example of using it**
    - ```
      #include <auto_ptr>
      void foo()
      {
          auto_ptr<Thing> p = new Thing;
          /* do stuff using p like a pointer */
      }   // object p points to is automatically deleted
      ```
  - **auto_ptr is Standard, but safe only to use in this exact way. Dangerous for other uses because of kinky ownership concept.**
    - *Ownership - who "owns" the resource? Who is responsible for getting rid of it when it is no longer needed?*
      - Other parties might be interested, but the ownership has to be clear to avoid confusion
        - Can't tolerate other parties using a "dead" resource, or trying to deallocate something already deallocated
    - *std::auto_ptr<> implements a strict ownership model - only one auto_ptr can point to an object at a time.*
      - if copied or assigned ownership goes to the new or left-hand-side auto_ptr
        - original or right-hand side is MODIFIED as a result of being copied
          - usually assignment operator and copy constructor have rhs or original object by CONST REFERENCE
          - but for this to work, original or r.h.s. object is by reference only, NOT const reference!
        - WIERD VIOLATION of normal language semantics
        - doesn't happen with ints or even ordinary pointers ...
      - if owning auto_ptr goes out of scope, pointed to object is deleted, even if original is still in scope - OOPS!
        - ```
          #include <auto_ptr>
          void goo(auto_ptr<Thing> p2)
          {
              p2->transmogrify();
          }   // what happens here?

          void foo()
          {
              auto_ptr<Thing> p1 = new Thing;
              p1->defrangulate();
              goo(p1);
              p1->deftrangulate();    // ???
          }
          ```
    - *If used for class member variable, be sure copy and assignment are properly defined! (need to be anyway)*
    - *If used elsewhere be very careful: don't use in function calls, return values*
    - *don't use in standard containers at all*
      - remember inserted values are COPIED in, and might be assigned around in the container ...
      - modern implementations will disallow - by requiring const reference in assignment and copy
    - *Can't use auto_ptr with an array either, which is where you are most likely to need memory management*
      - does delete, not delete[] of pointed-to object.
      - but still, use std::vector<> or some other container instead!

- **auto_ptr is in the Standard Library primarily because it was the first idea of a smart pointer that everybody could agree on, but it isn't very useful. I don't recommend trying to use it except for the rare simple case shown in the example.**

- **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 tr1::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 consistent with the conceptual constness of the object - merely referring
      to an object with a Smart_Pointer should not for 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
```

- **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
*/
```

- **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 get's executed - not very automatic!*
  - **Better way to fix - weak pointers:**
    - *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 boost/tr1 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 observed 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 status.
    - *In tr1, 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 ref count object can delete itself.*
    - *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.*
- *At this point, the boost implementation is the best compromise around.*
  - www.boost.org—smart_ptr.htm
  - **www.boost.org—weak_ptr.htm**
  - part fo the Std. Lib. Technical Report No. 1 - scheduled to become part of the Standard

- **Summary of boost/tr1 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 boost/tr1 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:**
    - www.boost.org—enable_shared_from_this.html
    - Clumsy to do stand-alone, but tr1 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!
    - Two Gotcha's
      - *Watch out!* If you have to write oopy constructor and assignment operator for the class, you have to make sure the weak pointer member gets copied and assigned also. The class enable_shared_from_this<> has copy constructor and assignment operators defined for this purpose. If you don't have to write your own copy ctor and operator=, then the compiler-supplied ones will correctly copy and assign the base class members for you.
      - *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 tr1::bad weak_ptr exception.

- **Demo of boost/tr1 shared_ptr, also shows cycle problem**
  - **Note implementation-specific access to the TR1 classes**
    - /*
      Demonstration of shared_ptr.
      This demo shows a how shared_ptr works in various situations with and without a cycle problem.
      This code assumes gcc 4.x with its version of Boost's TR1.
      */

      #include <iostream>
      #include <tr1/memory> // glibstd++ implementation of tr1

      using namespace std;
      using namespace std::tr1; // tr1 namespace is nested in std namespace

      /* 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 boost/tr1 weak_ptr, showing solution to cycle problem**
  - **Note implementation-specific access to the TR1 classes**
    - /*
      Demonstration of shared_ptr and weak_ptr, showing how cycle problems can be solved.
      This code assumes gcc 4.x with its version of Boost's TR1.
      */
      #include <iostream>
      #include <tr1/memory> // glibstd++ implementation of tr1

      using namespace std;
      using namespace std::tr1; // tr1 namespace is nested in std namespace

      /* 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 boost/tr1 shared_from_this()**
  - **Based on previous examples**
  - /*
    Demonstration of shared_ptr's shared_from_this facility.
    This code assumes gcc 4.x with its version of Boost's TR1.
    */

    #include <iostream>
    #include <tr1/memory> // glibstd++ implementation of tr1

    using namespace std;
    // without this using statement, we have to write std::tr1::shared_ptr, etc
    using namespace std::tr1; // tr1 namespace is nested in std namespace

    /* 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 std::tr1::enaable_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
*/
```