
CS11 *Advanced C++*

Fall 2009-2010

Lecture 5

Today's Topics

- Using **make** for build automation
 - Using **doxygen** to generate API docs
 - C++ exception handling
 - “Resource Allocation Is Initialization” pattern
 - Smart pointers!
-

Build Automation

- Standard development cycle:
 - Write more code
 - Compile
 - Test
 - Repeat until done...
 - Automating this process saves lots of time
 - Intelligent build tools *dramatically* improve build-times for large software projects
 - Projects that take minutes or hours to build
-

make

- **make** is a standard tool for automating builds
 - Command-line utility, very ubiquitous!
 - Takes input files and produces output files, based on a “makefile”
 - Several versions of **make**: GNU, BSD, ...
 - **make** is largely used for C and C++ projects
 - Sometimes other build tools are used for C/C++
 - Perforce Jam is one common alternative
 - Visual C++ provides **nmake** command-line build program
 - Other languages typically have their own build tools
 - e.g. Ant is frequently used for Java projects
-

Other Build-Tools

- For large projects, other tools are also used alongside **make**
 - **autoconf**, etc. for providing source-portability
 - Generates a **configure** script for configuring your program

```
./configure --prefix = /usr/local/myprog
```

```
make
```

```
make install
```
 - **libtool**, etc. for building shared libraries on different platforms
 - **install**, **rpmbuild**, etc. for updating system directories
 - **doxygen**, **DocBook**, etc. for doc-generation
-

Makefiles

- The makefile describes build targets
 - Each target specifies its dependencies
 - Each target also specifies how to build that target from the dependencies
 - Typical filename is **Makefile** or **makefile**
 - **make** looks for these by default
 - Preferred name is **Makefile**
 - Can specify another makefile using **-f** option
-

Makefiles (2)

- When **make** is run, a build target can be specified
 - **make raytrace**
 - If no target is specified, the first target in the makefile is run
 - **make**
 - First target is usually named **a11**, and builds everything
 - Can also specify multiple build targets:
 - **make clean raytrace**
 - Whitespace matters in makefiles!
 - Indentation is significant!
 - Tabs must be used for indentation!
-

Example Makefile

- Form of rules:

```
target : dependencies
        commands
```

- Example:

```
lab4 : main.o entry.o except.o config.o
      g++ -o lab4 main.o entry.o except.o config.o
```

```
main.o : main.cc entry.hh except.hh config.hh
      g++ -Wall -c main.cc
```

```
... (more rules)
```

```
clean :
      rm -f lab4 *.o *~
```

- Lines indented with tab characters – spaces won't work!
 - A line can be continued on next line, by ending it with \
 - Can specify multiple commands, if rules are separated by a blank line
-

Real Build Targets

- From our example:

```
main.o : main.cc entry.hh except.hh config.hh
        g++ -Wall -c main.cc
```

- In this case, `main.o` is a real file
 - **make** will only build what is *needed*
 - If target file's date is older than any dependency, **make** will rebuild the target
 - To force a file to be rebuilt, **touch** it:
 - `touch main.cc`
 - Sets file's modification-time to current system time
 - Touching a nonexistent file will create a new empty file
-

Phony Build Targets

- From our example:

```
clean :
```

```
    rm -f lab4 *.o *~
```

- In this case, **clean** is *not* a real file
 - What if there happened to be a file named **clean** ?
 - ❑ Our rule wouldn't run!
 - ❑ **make** would see the “build-target” file, and assume it didn't have to run
 - Use **.PHONY** to say that **clean** target isn't a file

```
.PHONY : clean
```

 - ❑ Now if a file named **clean** exists, **make** ignores it
-

Chains of Build Rules

- **make** figures out the graph of dependencies

```
lab4 : main.o entry.o except.o config.o
      g++ -o lab4 main.o entry.o except.o config.o
```

- If any of **lab4**'s dependencies don't exist, **make** will use their build rules to make them

```
main.o : main.cc entry.hh except.hh config.hh
        g++ -Wall -c main.cc
```

- **make** will give up if:
 - A dependency can't be found, and there's no build rule that shows how to make it
-

Makefile Variables

- Makefiles can define variables

```
OBJS = main.o entry.o except.o config.o
```

- Can use variables in build rules

```
lab4 : $(OBJS)
```

```
    g++ $(OBJS) -o lab4
```

- `$(var-name)` tells `make` to expand the variable

- Use variables to avoid listing the same things all over the place

- Same as code reuse: only make changes in one place

- Makefile variable names are usually **ALL_CAPS**

Implicit Build Rules

- **make** already knows how to build certain targets
 - Those targets have built-in rules for building them
 - These built-in rules are called *implicit* build rules
 - Example:
 - A makefile has `main.o` as a dependency, but no build rule
 - If `main.c` exists, **make** will use `gcc` to generate `main.o`
 - If `main.cc` exists, **make** will use `g++` to generate `main.o`
 - **make** has quite a few implicit build rules
 - Read **make** documentation for more details!
-

Using Implicit Rules

- Implicit rules make your makefiles *much* shorter

- Can leave out rules for all the object files

```
OBJS = main.o entry.o except.o config.o
```

```
lab4 : $(OBJS)
      g++ -o lab4 $(OBJS)
```

```
clean :
      rm -f lab4 *.o *~
```

```
.PHONY : clean
```

- What about header file dependencies?

- Can specify rules for each object file:

```
main.o : entry.hh except.hh config.hh
```

- No command – these rules just specify dependencies

- **makedepend** auto-generates these from your source files!
-

Definitions of Implicit Rules

- Examples of implicit rules:

```
# C compilation implicit rule
```

```
%.o : %.c
```

```
$(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@
```

```
# C++ compilation implicit rule
```

```
%.o : %.cc
```

```
$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@
```

- Variables are used for compiler and options!

- ❑ **CC** is C compiler, **CXX** is C++ compiler
 - ❑ **CFLAGS**, **CXXFLAGS** are compiler-options
 - ❑ **CPPFLAGS** are the preprocessor flags
 - ❑ Default values are for **gcc** and **g++**
 - ❑ Can easily customize these rules, by setting these variables at the top of your makefile
-

Definitions of Implicit Rules

- Implicit rules use patterns:

```
# C compilation implicit rule
```

```
%.o : %.c
```

```
$(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@
```

```
# C++ compilation implicit rule
```

```
%.o : %.cc
```

```
$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@
```

- Special syntax for pattern-matching

- % matches the filename
- \$< is the first prerequisite in the dependency list
- \$@ is the filename of the target
- These \$... values are called automatic variables
- Other automatic variables too!
 - e.g. \$^ is list of *all* prerequisites in the dependency list

make References

- For more details, see the GNU **make** manual
 - <http://www.gnu.org/software/make/manual/>



Automatic Document Generation

- Automating API-doc generation is a very powerful technique
 - Comment your code according to a specified style
 - Run a documentation-generator on your code
 - Produces API documentation of your code, in HTML, PDF, etc. formats, ready for distribution!
 - The documentation is in one place – your source
 - Tools can use the code as well as your comments in the generated output
 - Several different options for doc-generation
 - We will use doxygen: <http://www.doxygen.org>
-

Doxygen Configuration

- Doxygen is driven by a config file
 - It will generate a template file for you:
 - `doxygen -g [filename]`
 - Default filename is `Doxyfile`
 - Should probably use a more descriptive name...
 - Customize the config file for your project
 - Set different configuration parameters as needed
 - Parameters are *well documented* in the config file
 - Parameter names are **ALL_CAPS**
 - (just like makefile variables)
 - Parameter-value can extend to next line, if current line ends with \ (backslash) character
 - Switches are specified with **YES** or **NO**
-

Doxygen Config Tips

- You should set:
 - ❑ **INPUT** (input files/directories)
 - ❑ **OUTPUT_DIRECTORY** (where results go)
 - ❑ **PROJECT_NAME**
 - Other good settings to use:
 - ❑ **JAVADOC_AUTOBRIEF = YES**
 - ❑ **EXTRACT_ALL = YES**
 - ❑ **EXTRACT_PRIVATE = YES**
 - ❑ **EXTRACT_STATIC = YES**
-

Commenting Your Code

- Several different formats are recognized

```
/**
 * This is a comment for my class.  It is spiffy.
 */
class MyClass { ... };
```

- `/**` starts the comment (javadoc style)
 - Can also start with `/*!` (Qt style)
 - Also several other options (see doxygen manual)
 - Classes, types, functions have a brief comment, and a detailed comment
 - If `JAVADOC_AUTOBRIEF` is defined in doxygen config, first sentence is used as brief comment.
 - Otherwise, must use `\brief` keyword in your comments
-

Structural Commands

- “Structural commands” specify what a comment is associated with
 - ❑ “This is a comment for the source file.”
 - ❑ “This is a comment for class **C**.”
 - ❑ “This is a comment for parameter **x** of the function.”
 - ❑ etc.
 - ❑ Allows Doxygen comments to be separated from entities that are being commented. (Not always recommended...)
 - Two different formats for structural commands
 - ❑ Doxygen format: `\cmd`
 - ❑ Javadoc format: `@cmd`
 - ❑ Can use either format, but be consistent! 😊
-

What Can Be Commented?

- Files can be given comments
 - Must do this for doxygen to pick up certain comments
 - Examples:
 - `/*! \file ... */` (Qt/Doxygen format)
 - `/** @file ... */` (Javadoc format)
 - Any type can be given a doxygen comment
 - Classes, structs, enums, typedefs, unions, namespaces
 - Comment should immediately precede the type
 - ...unless you are using structural commands
 - Preprocessor definitions can also be commented!
 - `#define` symbols, macros
-

Commenting Variables and Functions

- Global/static variables, and member variables
 - Comments can precede the variable:

```
/** My special widget. */  
SpecialWidget sw;
```
 - Or they can follow the variable, on the same line:

```
SpecialWidget sw; /**< My special widget. */
```

 - (Note the < character)
 - Functions and their parameters/return values
 - Parameters follow this pattern:
 - @param name Description
 - \param name Description
 - Return value is documented with \return or @return
-

Running **doxygen**

- Doxygen is simple to run:

```
doxygen [filename]
```

- **doxygen** uses **Doxyfile** if no config file is given
 - Basically no command-line arguments
 - Config file contains all the details!
 - Results are stored in output directory
 - Each format gets its own subdirectory
 - **html** for HTML output, **latex** for LaTeX, etc.
 - Can specify alternate output directories if desired.
-

doxygen References

- For more details, see the **doxygen** manual
 - <http://www.stack.nl/~dimitri/doxygen/manual.html>
 - <http://www.doxygen.org>



C++ Exceptions

- Exceptions are nice for reporting many errors
 - Code throwing the exception can detect the problem, but doesn't know how to handle it.
 - Code that catches the exception knows what to do about the problem.
 - With *careful* implementation of constructors and destructors, resources get cleaned up, too.
 - C++ Standard Library provides a number of standard exception classes
-

Simple Example

```
void calculate(float x) {
    if (x < 0)
        throw domain_error("x is negative");

    // Do our calculation.
    ...
}

int main() {
    float x;
    cin >> x;
    try {
        calculate(x);
    } catch (domain_error) {
        cout << "Caught a domain error!" << endl;
    }
}
```

What Happened?

```
void calculate(float x) {
    if (x < 0)
        throw domain_error("x is negative");

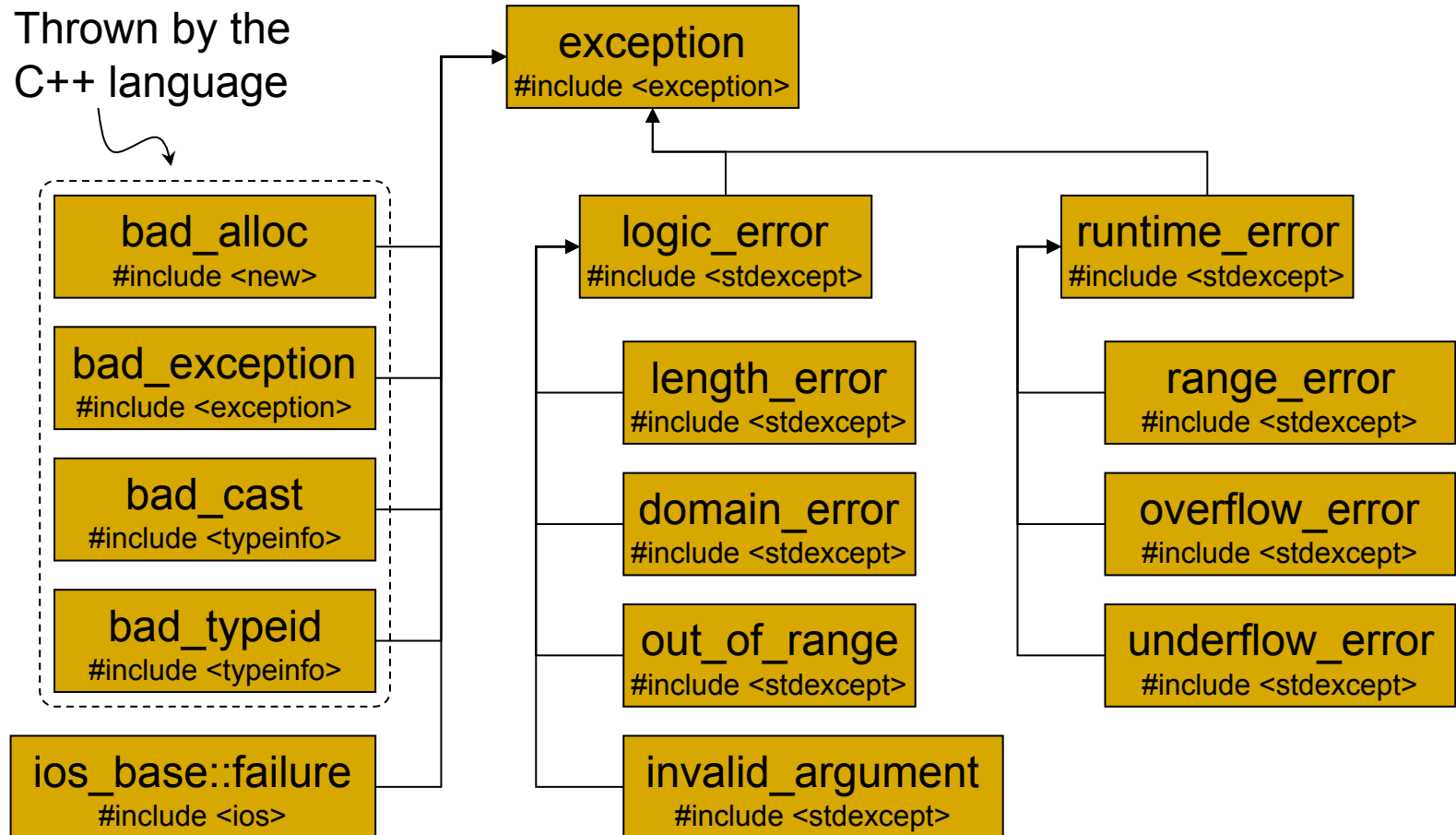
    // Do our calculation.
    ...
}

int main() {
    float x;
    cin >> x;
    try {
        calculate(x);
    } catch (domain_error &de) {
        cout << "Caught a domain error!" << endl;
        cout << de.what() << endl;
    }
}
```

C++ Standard
Exceptions provide a
`what()` function to
retrieve error details.

C++ Standard Exceptions

Thrown by the
C++ language



Standard Exception Hierarchy

- Two major kinds of standard exceptions
 - **logic_error**
 - ❑ Intended for “preventable” errors, a.k.a. bugs
 - ❑ Invalid function arguments, violated invariants, etc.
 - ❑ A potential alternative to using **assert()**
 - ❑ Steer clear of these for reporting runtime errors!
 - **runtime_error**
 - ❑ “All other errors.”
 - ❑ Errors that can only be detected as the program executes
-

Using the Standard Exceptions

- These are available for use in your programs.
 - Use them as-is, subclass them, or ignore them and make your own!
 - You will *probably* have to handle them, at least...

“Some people view this as a useful framework for all errors and exceptions; I don’t.”

- Bjarne Stroustrup

The C++ Programming Language §14.10

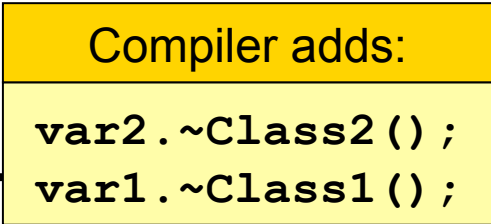
Local Variables and Destructors

- Normally, destructors are called when variables go out of scope

```
void myFunction() {  
    Class1 var1;  
    Class2 var2("out.txt");  
  
    var1.doStuff(var2);  
}
```

Compiler adds:

```
var2.~Class2();  
var1.~Class1();
```



- Compiler inserts destructor calls into the appropriate places
 - Your code doesn't manually call destructors, ever.
-

Destructors and Exceptions

```
void myFunction() {  
    Class1 var1;  
    Class2 var2("out.txt");  
  
    var1.doStuff(var2);  
  
    var2.~Class2();  
    var1.~Class1();  
}
```

THROW!

cleanup
(stack unwinding)

propagate
exception

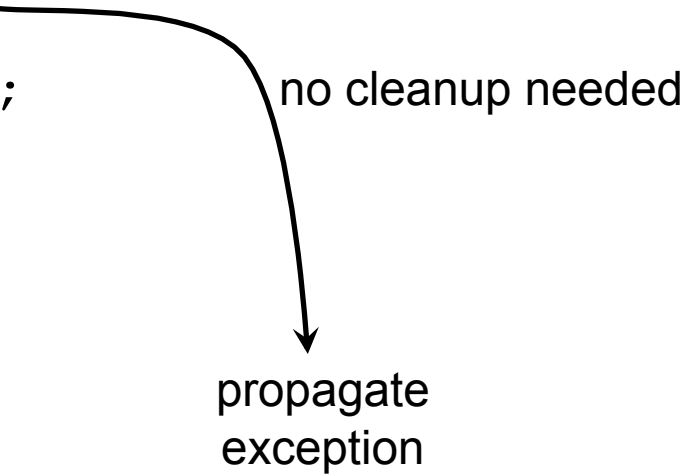
- What happens if **var2** constructor throws?
 - Only **var1** was constructed, so only **var1** destructor gets called

Destructors and Exceptions (2)

```
void myFunction() {  
    Class1 var1; THROW!  
    Class2 var2("out.txt");  
  
    var1.doStuff(var2);  
    var2.~Class2();  
    var1.~Class1();  
}
```

no cleanup needed

propagate exception



- What happens if **var1** constructor throws?
 - Nothing was constructed, so no destructors get called

Destructors and Exceptions (3)

```
void myFunction() {  
    Class1 var1;  
    Class2 var2("out.txt");  
  
    var1.doStuff(var2); THROW!  
  
    var2.~Class2();  
    var1.~Class1();  
}
```

cleanup
(stack unwinding)

propagate
exception

- What happens if `var1.doStuff(var2)` throws?
 - Both `var1` and `var2` were constructed, so both destructors get called (in reverse order of construction)

Classes and Exceptions

- Similar model used when constructors throw

```
class Logger {  
    LogConfig config;  
    RotatingFile outputFile;  
public:  
    Logger(const string &configFile) {  
        ... // initialize logger THROW!  
    }  
    ...  
};
```

- What happens if the constructor body throws?
 - The new `Logger` instance failed to be constructed
 - `config` and `outputFile` have already been initialized, so their destructors are automatically called
-

Classes and Exceptions (2)

- Member initialization might also throw

```
class Logger {
    LogConfig config;
    RotatingFile outputFile;
public:
    Logger(const string &configFile) :
        config(configFile), outputFile(config)
    {
        ... // initialize logger
    }
    ...
};
```

THROW!



- What happens if `outputFile` constructor throws?
 - The new `Logger` instance failed to be constructed (again)
 - `config` was already initialized, so its destructor gets called

Classes and Exceptions (3)

- Another member constructor throws

```
class Logger {
    LogConfig config;
    RotatingFile outputFile;
public:
    Logger(const string &configFile) :
        config(configFile), outputFile(config)
    {
        ... // initialize logger
    }
    ...
};
```

THROW! →

- What happens if `config` constructor throws?
 - The new `Logger` instance failed to be constructed (yet again)
 - Nothing was initialized, so no member destructors are called

Heap-Allocation and Exceptions

- What if members are heap-allocated?

```
Simulator::Simulator(SimConfig *pConf) {  
    // Initialize my simulator members.  
    entityData = new Entity[pConf->maxEntities];  
    playerData = new Player[pConf->maxPlayers];  
}
```

- If an allocation fails, **new** will throw **bad_alloc**
- What happens if second allocation throws **bad_alloc**?
 - Simple: **entityData** doesn't get cleaned up



A Safer Constructor

- Can fix the problem by doing this:

```
Simulator::Simulator(SimConfig *pConf) :
    entityData(0), playerData(0)
{
    try {
        entityData = new Entity[pConf->maxEntities];
        playerData = new Player[pConf->maxPlayers];
    }
    catch (bad_alloc &ba) {
        delete[] entityData;
        delete[] playerData;
        throw;    // Don't forget to propagate this!
    }
}
```

- Not the prettiest code, but at least it's safe.
-

Again and Again!

- This pattern gets old fast:

```
void initSimulation() {  
    SimConfig *pConf = new SimConfig("sim.conf");  
    Simulator *pSim = new Simulator(pConf);  
    ...  
}
```

- What if **Simulator** constructor throws?
 - (sigh)

- This approach to leak-free, exception-safe code is a pain!
-

Safe Dynamic-Resource Management

- Problem:

- Dynamic allocation of resources, plus exception-handling, is a potentially dangerous mix!
 - Memory allocated with `new`
 - Opening files, pipes, etc.
 - Threads, mutexes, condition variables, semaphores, ...
 - Just catching exceptions isn't enough!
 - Also need to release any resources that were allocated before the exception was thrown.
-

Typical Resource Allocation Model

- General form of the problem:

```
void doStuff() {  
    // acquire resource 1  
    // ...  
    // acquire resource N  
  
    // use the resources  
  
    // release resource N  
    // ...  
    // release resource 1  
}
```

- Resources usually released in opposite order of allocation
 - Hey, C++ constructors and destructors do this!
 - Local variables are created and destroyed this way
-

Easier Leak-Proofing Approach!

- Make a wrapper class for managing a dynamic resource
 - Constructor allocates the dynamic resource
 - (or constructor assumes ownership of the resource)
 - Destructor frees the resource
 - Use the wrapper class for local variables
 - (Otherwise, you're back to the old problems again...)
 - “Clean up” handlers become unnecessary
 - When exception is thrown, C++ calls wrapper-class destructor automatically, since it's local.
-

“Resource Allocation Is Initialization”

- This pattern is called “Resource allocation is initialization.”
 - A local variable’s constructor immediately assumes ownership of the dynamic resource
 - C++ will call the destructor at the Right Time.
 - Typically realized as “smart pointers”
 - They follow this model for heap-allocated memory
 - This can be applied to *any* dynamic resource
 - Files! Semaphores! Mutexes! ...
-

The C++ `auto_ptr` Template

- C++ Standard Library's contribution to memory management tools
 - A simple smart pointer, in `std` namespace
 - Implements “resource allocation is initialization” pattern for object-pointers.
 - Great tool, *when used correctly!*
 - Must understand what `auto_ptr` does!
 - It is a single-ownership smart pointer
 - Only one `auto_ptr` instance owns a given object-pointer.
-

Implications of Single-Ownership

- Assigning one `auto_ptr` to another, changes what is assigned.

```
auto_ptr<Widget> spw(new Widget());  
auto_ptr<Widget> spw2;  
...  
spw2 = spw; // Now spw == 0!
```

- Copying one `auto_ptr` into another (e.g. with copy-constructor), changes what is copied.

```
auto_ptr<Widget> spw(new Widget());  
...  
auto_ptr<Widget> spw2(spw); // Now spw == 0!
```

- Makes sense in context of single-ownership idea
- Will catch you off guard if you don't know this!
 - Copying or assigning something doesn't usually change it

STL Containers and `auto_ptr`

- STL containers and `auto_ptr` don't mix!
 - Copy operations on STL container-items should not change what is copied
 - `auto_ptr` has a nonstandard meaning for copying and assignment
 - Don't ever use `auto_ptr` to wrap items in STL containers. It is forbidden.
 - Unfortunately, some (bad) STL implementations will allow it...
-

Smart-Pointer Operator Overloads

- Smart pointer classes overload these operators:
 - Dereference (member-access) operator \rightarrow
 - Dereference operator $*$ (not multiplication...)
 - Allows the smart-pointer to act like a pointer
 - STL iterators do basically the same thing...
 - Slightly different use cases
 - Smart pointers don't support pointer-arithmetic
 - For example: no $++$ and $--$ operators!
-

A Smart-Pointer Sketch

- A simple (and very incomplete) example:

```
template<typename T>
class PointerToT {
    // Pointer to the resource being managed
    T *ptrT;
public:
    PointerToT(T *p) : ptrT(p) { }
    ~PointerToT() { delete ptrT; }
    T * operator->() { return ptrT; }
    T & operator*() { return *ptrT; }
    ...
};
```

- The hard parts:
 - Define semantics of copy-construction, assignment, casting operations up/down class hierarchies, etc.
-

The Boost Libraries

- Boost is a collection of portable, peer-reviewed libraries
 - Some libraries in process of being added to C++ Standard
 - Some Boost developers are C++ Standards Committee members
 - Freely available, good license
 - A bit of a pain to set up
 - Uses Boost.Jam build tool (derived from Perforce Jam)
 - Some templates have binaries to build and link against
 - Can exclude libraries with binaries (if you config it just right)
 - Or, can use some Boost headers without building it all...
-

Boost `shared_ptr` Template

- In Boost's smart pointer library
- `shared_ptr` supports multiple-ownership
 - Uses reference-counting to know when to delete
 - Uses templates and operator overloading to make use seamless.

```
// Use a typedef to avoid unnecessary typing
typedef boost::shared_ptr<Widget> SPWidget;
```

```
vector<SPWidget> widgetSPs;
SPWidget wsp(new Widget(...));
```

```
cout << wsp->getID();           // Can treat smart-pointer
Widget w = *wsp;                // like a normal Widget*
```

```
widgetSPs.push_back(wsp);      // No need to clean up!
```

Shared Pointers: The Good

- Clean up is automatic!
 - No cleanup exception handlers
 - No looping over collections of pointers, to delete the contents
 - No delete functors
 - If you assign the result of every **new** to a suitably clever, *named* smart-pointer variable:
 - No need for explicit **delete** code
 - The need for **try/catch** blocks will be rare
-

Shared Pointers: The Bad

- Can *still* have subtle bugs. (Thanks, C++!)

- This works fine:

```
vector<SPWidget> widgetSPs; // Vector of shared_ptrs
widgetSPs.push_back(SPWidget(new Widget()));
```

- This *might* leak:

```
processWidget(SPWidget(new Widget()), getTicket());
```

- No guarantee on order of evaluation in function arguments!
 - New `Widget` is constructed...
 - ...then `getTicket()` goes and throws an exception!
 - `shared_ptr` constructor never got called! Leaky...
 - Moral: “Avoid using unnamed `shared_ptr` temporaries to save typing...”
-

Final Pointer Tips

- **auto_ptr** is a nice tool, when used properly
 - Remember that it's a single-ownership pointer
 - Good for heap-allocated memory used within one function
 - Never use it inside STL containers
 - Use Boost's **shared_ptr** for multiple-ownership smart pointers
 - Remember other smart pointer pitfalls, too
 - Anonymous smart pointers in function args, for example
 - See the Boost smart-pointer documentation
 - http://www.boost.org/libs/smart_ptr/smart_ptr.htm
 - Effective STL by Scott Myers
-

This Week's Assignment

- An easy one, for once... 😊
 - Create a **makefile** for your program
 - Update your source code to use Boost smart pointers
 - Update your source code to use doxygen-style comments
 - Add a **docs** target that runs doxygen
 - Make sure your commenting is complete!
 - Every class, member function, data-member, etc.
-