# Polishing Up Your Project

## CS11 Advanced C++ Lab 5

This week's lab isn't focused so much on implementation, but rather on adding some tools and components that will make your project much more reliable and useful. This includes incorporating smart pointers to manage dynamically allocated resources, using an automated API-doc generator to produce browsable documentation from your source, and using a *makefile* to automate the entire process of compilation.

## Build Automation

This week you will get your program building using a `Makefile`. Feel free to use implicit build rules, as shown in class. Your makefile should satisfy the following requirements, at a minimum:

- The default target (if you just type `make` with no arguments) should build the ray tracer binary.
- There should be a `clean` target for deleting build files. *Make a backup of your work before testing this rule, just in case!*
- Make sure to specify which targets are "phony" in your makefile!
- Create a `depend` target that uses the `makedepend` utility to update your `Makefile` with header file dependency details. You can type `man makedepend` for more details. (To use `man`: the arrow keys scroll up and down, Space scrolls down a whole page, "b" scrolls up a page, and "q" exits.)

  Before testing this build rule, make a backup of your `Makefile`! The `makedepend` utility actually modifies your `Makefile`, so it's important to make sure you know how it does that before you unleash it.

Once you have your `Makefile` all written and working properly, don't forget to add it to the Subversion repository using `svn add`, and then `svn commit`. Remember that the file doesn't actually get stored into the repository until you issue that last commit command. And of course, specify a good log message when adding the file. It doesn't necessarily have to be long, but it definitely needs to be complete.

At this point in your project, you might also find it helpful to create an "image" target that runs your raytracer to generate an image. This makes it easier to test your program, for the time being. You can make the target depend on your raytracer's binary name, so that it will be automatically rebuilt when any of the sources change.

## Boost Smart Pointers

So far your ray tracer is using STL collections to store scene objects and lights, but you are storing pointers to these objects because of STL's tendency to copy the contents of containers. Since our scene-objects form a class hierarchy, we can run into the problem of slicing if we aren't careful to store pointers instead of the objects themselves.

Of course, because of this, you also have to make sure that the dynamically allocated objects are freed at the proper time, and this can be complicated even more by the addition of exception handling. This is where *smart pointers* can become a very powerful tool. The Boost library, in particular, provides a `shared_ptr` template that implements a multiple-ownership smart pointer that is nearly perfect for use in STL containers.

Update your program to manage all dynamically allocated objects using the `boost::shared_ptr` template class. You can `#include "boost/shared_ptr.hpp"` at the top of your source files, and you can use the Boost libraries on the CS cluster:

```
BOOST_INC = /cs/courses/cs11/resource/advcpp/include

CPPFLAGS = -I$(BOOST_INC) ...
```

Since Boost may be installed in different places on different systems, it's *strongly* recommended to use a Makefile variable that can be easily updated in one place.

Also, use `typedef`s in your source code to make the use of the shared pointers a little easier to work with in your code. You might want to declare a special typedef for each of the objects that you dynamically allocate, for example:

```
// Boost shared-ownership smart pointer for scene objects.
typedef boost::shared_ptr<SceneObject> SPSceneObject;
```

You really should only declare smart-pointers for the pointer-types that you use in your program; otherwise you can get into some trouble with casting them. (More on this next week...) For now, only use shared-pointers for the types of pointers you store in your STL collections. For example, you would have a `SPSceneObject` shared-pointer, but you wouldn't have a `SPSphere` or `SPPlane` because you don't need those anywhere.

Also, you should assign each dynamically-allocated object to a smart pointer right away, to avoid the possibility of unexpected memory leaks. Your "scene" class has a few member-functions that take pointers to scene-objects and lights; change these functions to take a smart-pointer to the object instead. These can be passed by-value since they are lightweight objects.

## API Documentation using Doxygen

Automated document-generation tools are very powerful and effective ways of generating professional looking documents for both developer use, and for end-user manuals. The cost is not very high; typically, the only requirement is to learn the commenting markup that the tool requires, and then the rest of the process is quite painless. The outputs of these tools are typically all ready for distribution, and only need to be packaged up.

One such open-source documentation system is called doxygen. It supports cross-platform, cross-language documentation generation, and it provides a good mechanism to encourage you to document your code as well.

1. Go through your project's source code and update all comments to use a format that doxygen can recognize and process. Ideally, you have already been commenting your source code very well to this point, but just in case you haven't, you can remedy that deficiency now, too.

   Focus on providing concise, clear specifications for each class, data member, and member function in your implementation. *Don't focus on the number of words* - make your statements count. In general, people will tend to ignore comments if they are too verbose, so the best tactic is to be brief, yet complete. (Most programmers err on the side of being brief and incomplete. Don't do that either!)

   Also think about what your comments' first sentence says. This is used as the "brief" documentation (when `JAVADOC_AUTOBRIEF` is turned on), and this is all that people will see when they are browsing the indexes. Give a good summary statement in that first sentence!

2. Configure and use doxygen to generate HTML API documentation from your comments. Remember that doxygen will generate a template configuration file for you, if you type `doxygen -g` *`filename`* at the command-prompt.

3. Add a `docs` target to your makefile to produce the auto-generated documentation. Have doxygen put its documentation in a `docs` subdirectory of your lab. (That will be `./docs` to your makefile.)

   Also add commands to your makefile's `clean` target, to delete the auto-generated documentation when someone runs that target.

4. Make sure to add your new doxygen config file to the Subversion repository for your project. However, ***don't*** add the new documentation directory to Subversion, because this documentation is generated by the build process.

   Instead, you can tell Subversion to ignore the documentation directory using Subversion properties. There is a special property called `svn:ignore` which lists the files that Subversion should ignore. You can specify this on a per-directory level. So, go into the parent directory where the `docs` directory is created, and type:

   ```
   svn propedit svn:ignore .
   ```

   This will start a text editor (again, whatever your `EDITOR` environment-variable specifies) for editing the property. Each line lists a file or subdirectory that Subversion should ignore. You can add a line `docs` to the property, and then exit the editor. Once you do this, Subversion will know to ignore this subdirectory when updating the working copy.

## All Done...

When you are finished, leave your work in your `~/cs11/advcpp/lab5` directory. Make sure to update your Subversion repository as well.

---


Last updated on February 13, 2008