CS11 Advanced C++

Spring 2012-2013

Lecture 4

Source Code Management

- You are working on a large software project…
- Problem 1: You break the code
  - Need to roll back to a previous version that works
- Problem 2: Other people also working on project
  - …perhaps on the *exact* same source files
- Problem 3: Centralized source of project info?
  - Maybe a website that shows current test pass-rate, most recent API docs, etc.
- A source code management system can solve all of these problems, and many more

Managing the Source Code

- Basic idea:
  - Store all project files in a repository
  - Repository keeps track of all changes to any file
  - Copies of the project are “checked out” from the repository
  - Developers are isolated from others’ changes
  - Changes to project files are “checked in” or “committed” back to the repository, when ready.
  - Multiple changes to the same file are merged
    - Automatically, if possible; otherwise, manually!

Distributed Version Control

- A new trend in version control systems:
  - Don’t use a central repository server!
- Distributed version control systems
  - Each user has a local repository
  - Users work against their own local repository
    - Check out a working copy, make edits, then check in
  - Users can synchronize with other repositories very easily
- Great for widely distributed software development
  - Open-source software, for example
- Used less often in commercial development teams
  - Software companies prefer to have a single central server
  - Can still use DVCS in a centralized manner, though

Version Control Systems

- Commercial centralized version control systems:
  - Perforce, Visual SourceSafe, BitKeeper, …
- Open-source centralized version control systems:
  - Subversion – written as a replacement for CVS
- Open-source distributed version control systems:
  - Git – written by Linus Torvalds
    - Used for Linux kernel development, Eclipse, PostgreSQL, …
  - Mercurial (**`hg`**) – distributed VCS written in Python
    - Used by Python project, vim, OpenOffice, GNU Octave, …
  - Bazaar – also written in Python
    - Used by Ubuntu project, GNU Emacs, MySQL, …

Using Subversion

- Two main commands in Subversion:
  - **`svn`**
    - Program used by developers to access the repository
    - Can check out files, check in, move, delete, etc.
  - **`svnadmin`**
    - The repository administration tool
    - Used rarely, by repository administrator
- Both programs take commands
  - Example: **`svn checkout ...`**
  - Both have a help command:
    - **`svn help`** or **`svn help`** ***`command`***

Setting Up a Repository

- Start by creating a repository
  - Repository contains all the config and data files
  - Command:

    `svnadmin create /path/to/repository`
  - Can be an absolute or relative path
- Can create your repository on the CS cluster

  `svnadmin create ~/cs11/advcpp/svnrepo`
- Subversion can use different storage layers
  - Filesystem storage, or BerkeleyDB
  - Default is filesystem – use that!

Accessing the Repository

- Subversion uses URLs to refer to repositories
  - Supports access via HTTP, if needed
- For local access, use a **file://** URL
  - On CS cluster: **file:///home/user/cs11/advcpp/svnrepo**
- Subversion also supports remote access
  - **svn://…** URL for use of Subversion's server
  - Or, **svn+ssh://…** URL for accessing over SSH
- For accessing CS cluster repository remotely:
  svn+ssh://user@login.cs.caltech.edu/home/user/cs11/advcpp/svnrepo

Importing Source Code

- Need to import initial project source into repository
  - **`svn import`** does this
  - Recursively adds a whole directory tree to repository
- Lay out your repository in a reasonable way
  - Each project (or subproject) should have its own directory
  - Example ray tracer directories:
    - **`raytracer`**
    - **`raytracer/docs`**
    - **`raytracer/tests`**
    - **`raytracer/scenes`**
    - *etc.*
- Subversion lets you move files/directories later, too

Importing Source Code (2)

- Go to directory with your source files
  - Clean up **`.o`** files, etc. – don't want to import those
- Import the directory tree into the repository
  - Usually want to specify a subproject to use

```
svn import \
  file:///home/user/cs11/advcpp/svnrepo/raytracer
```

  - Subversion will add all files in the local directory (and subdirectories) into a **`raytracer`** subdirectory of your repository
  - Can also specify a path to directory to import

Working On Your Project

- Now, repository is the central store of all versions of all files
  - Can check out any version of any file
  - Usually want the most recent version to work with
- Retrieve a “working copy” of your project
  - A local copy of a particular version of the files
  - You can make changes in isolation
  - Can periodically sync up with other changes that have occurred
  - Once your local copy works properly, check it in!

Checking Out Files

- To check out files:
  - **`svn checkout`** ***`url`*** (or, **`svn co`** ***`url`***)
  - URL specifies both repository location, and directory within repository
- For example, to get raytracer project from your repository:

```
svn checkout \
  file:///home/user/cs11/advcpp/svnrepo/raytracer
```

  - Will create a local directory named **`raytracer`**, with project files in it
- To update local working copy:

  `svn update` (or, **`svn up`**)

  - If performed within working copy, no URL needed!

Working with Local Files

- Can add new files using **`add`** command
  - From within working copy:

    `svn add path1 path2 ...`
  - Can add whole directories
    - Subversion will recurse through directory contents
- Can delete files using **`delete`** command
  - Again, within working copy:

    `svn delete path1 path2 ...`
- Can move files using **`move`** command

  `svn move frompath topath`

Committing Changes

- Changes to working copy must be committed before they are visible to anyone else
  - Includes add/delete/move operations
- Subversion makes sure your local working copy is up to date first
  - Can't commit until you have latest version incorporated
- Issue **`commit`** command

  `svn commit`
  - Can specify files to commit, if desired
  - By default, commit operation is *recursive*

Commit Logs

- Subversion will prompt you for a commit log message
  - Describes changes you made in that particular commit
- Always give a descriptive commit message, even for small changes!
  - Other people need to know what you have done
  - You may need to remind yourself, too!
- Subversion client will start an editor for you
  - Can specify which editor to use with the **`SVN_EDITOR`** (or **`EDITOR`**) environment variable
  - For short messages, use the **`-m`** command-line option to specify the commit message

Discarding Changes

- Use **`svn revert`** to discard local changes
  - Subversion keeps a local copy of original files, so operation doesn't require actual repository access
  - Can't actually revert *every* local change
    - e.g. can't restore deleted directories
- Another option:
  - Simply delete working copy and fetch a new one
  - *Does* require repository access, so a little slower than using **`svn revert`**

Important Version-Control Tips!

- <u>Always</u> compile and test your code before checking it in
  - Your mistakes <u>will</u> affect other people badly.
  - Repository version of code should *always* compile, and ideally, work well too.
- Keep your working copy updated with latest version of repository code
  - Avoids big headaches from getting out of sync with other development progress

Subversion Documentation

- Subversion website:
  - http://subversion.tigris.org
- The Subversion Book (very useful!)
  - http://svnbook.red-bean.com
  - Subversion v1.6 available on CS cluster – use version of Subversion Book for that version
- Don’t forget **`svn help`** too

Iterators and `const`

- Iterators are non-const by default
- A class for managing a data-set of samples:

```
class DataSet {
    vector<float> samples;
public:
    ...
    int countValues(float val);
};
```

- `countValues()` really should be `const`
  - Just scans through the set of samples!
  - Calling `countValues()` doesn't change the data-set

Counting Values

- A version of **`countValues()`** that uses iterators:

```
int DataSet::countValues(float val) const {
    vector<float>::iterator iter;
    int total = 0;
    for (iter = samples.begin();
         iter != samples.end(); iter++) {
        if (*iter == val)
            total++;
    }
    return total;
}
```

  - Doesn't compile. ☹

```
error: invalid conversion from `const float* const' to `float*'
```

Counting Values and Preserving **const**

- To preserve const, use **`const_iterator`** instead:

```
int DataSet::countValues(float val) const {
    vector<float>::const_iterator iter;
    int total = 0;
    for (iter = samples.begin();
         iter != samples.end(); iter++) {
        if (*iter == val)
            total++;
    }
    return total;
}
```

  - Can't change collection's contents through a **`const_iterator`**

Counting Const Values with STL

- Another version of our function, using only STL:

```
int DataSet::countValues(float val) const {
    return count_if(samples.begin(), samples.end(),
                    bind2nd(equal_to<float>(), val));
}
```

- STL algorithms and functions work properly with **`const`**-correctness, automatically
  - …unless the algorithm changes the collection, of course! (e.g. **`sort`** or **`reverse`**)
- Another reason to prefer STL algorithms for working with STL containers, if possible

Subclassing Templates

- You write a template for a base-class:

```
template<typename T>
class Base {
public:
    void f() { }
};
```

- Then you write a template for a derived class:

```
template<typename T>
class Derived : public Base<T> {
public:
    void g() {
        f();
    }
};
```

- This code won't compile! ☹

Subclassing Templates (2)

- Inside **`Derived<T>::g()`**, the name **`f`** doesn't depend on the template-parameter **`T`**

```
template<typename T>
class Derived : public Base<T> {
public:
    void g() {
        f();
    }
};
```

  - Known as a "nondependent name"
- Compiler doesn't look in dependent base-classes when looking up nondependent names
  - i.e. compiler doesn't check **`Base<T>`** when looking for **`f`**

Subclassing Templates (3)

- Two options:
  - Okay: use `Base<T>::f()` instead of just `f()`
    - **Don't do this if `f` is virtual!** Might not get the right results.
  - Better: use `this->f()` instead of just `f()`
    - `this` is always *implicitly* dependent in a template
    - e.g. `this` has a type of `Derived<T>*` in this example
- Fixed version:

```
template<typename T>
class Derived : public Base<T> {
public:
    void g() {
        this->f();
    }
};
```

More Template Subclassing Fun

- You write this:

```
void f() { }        // A global function f

template<typename T> class Base {
public:
    void f() { }  // A different member function f
};
template<typename T> class Derived : public Base<T> {
public:
    void g() {
        f();
    }
};
```

- This code *does* compile! ☹

More Template Subclassing Fun (2)

- When compiler tries to resolve **`f`**, it searches the enclosing scope of **`Derived<T>`**
  - That scope contains the global function **`f`**
    - Qualified name of **`f`** is **`::f()`**
- The code compiles, and **`::f()`** is called instead of **`Base<T>::f()`**
- Moral:
  - Be very careful when subclassing templates!
  - Often need **`this->`** or **`Base<T>::`** for base-class member access, when deriving from a template

Even More Template Subclassing Fun

- You create a class-template for subclassing:

```
template<typename T>
class Base {
public:
    typedef T* ptr_t;  // type used by templates
};
```

- Use the base-template in a subclass where **`T`** = **`int`**:

```
class Derived : public Base<int> {
public:
    void f() {
        ptr_t pVal;  // pVal = pointer to int
        ...
    }
};
```

- Compiles fine. Works great. Yay for us.

Even More Template Subclassing Fun (2)

- Turn **`Derived`** into a class-template too:

```
template<typename T>
class Derived : public Base<T> {
public:
    void f() {
        ptr_t pVal;
        ...
    }
};
```

- Completely breaks.

```
error: `ptr_t' undeclared
```

- Change **`ptr_t`** to **`Base<T>::ptr_t`** ?
  - Still completely breaks. And, the error is weird:

```
error: expected `;' before "pVal"
```

Even More Template Subclassing Fun (3)

```
template<typename T>
class Derived : public Base<T> {
public:
    void f() {
        Base<T>::ptr_t pVal;
        ...
    }
};
```

- When template definition is parsed, compiler can't guess what the name **`Base<T>::ptr_t`** refers to
  - **`Base<T>`** hasn't been instantiated yet, when **`Derived<T>`** is parsed. Don't know yet what **`ptr_t`** will actually be!
    - Could be a variable, a function, a typedef, etc.
  - Compiler could *try* to infer from the context, but isn't smart enough. So, it makes an assumption instead.

Even More Template Subclassing Fun (4)

- C++ standard specifies that names within templates are assumed to be <u>non-types</u> by default.
- To specify that a name within a template refers to a type, must put **`typename`** in front of the name.

```
template<typename T>
class Derived : public Base<T> {
public:
    void f() {
        typename Base<T>::ptr_t pVal;
        ...
    }
};
```

  - Now the code compiles fine.
  - Tells the compiler that name **`Base<T>::ptr_t`** is a type

Even More Template Subclassing Fun (5)

- Can also redefine **`ptr_t`** inside of **`Derived<T>`**

```
template<typename T>
class Derived : public Base<T> {
public:
    typedef typename Base<T>::ptr_t ptr_t;

    void f() {
        ptr_t pVal;
        ...
    }
};
```

  - A little grungy, but saves a lot of typing.

Simple Templates and Types

- This issue exists even without template subclassing

```
template<typename T>
class DataSet {
    vector<T> samples;
public:
    int countValues(const T &val) const {
        vector<T>::const_iterator iter = ...
    }
};
```

- The **`vector<T> samples`** part is actually fine!
  - Compiler <u>knows</u> that **`vector<T>`** is the name of a class-template, when **`DataSet`** template is parsed
- **`vector<T>::const_iterator`** is the bad part
  - Don't know what **`const_iterator`** will be, until **`vector<T>`** is instantiated

Simple Templates and Types (2)

- Simple fix to the problem:

```
template<typename T>
class DataSet {
    vector<T> samples;
public:
    int countValues(const T &val) const {
        typename vector<T>::const_iterator iter = ...
    }
};
```

- Again, the STL version doesn't have this problem ☺
  - STL class- and function-templates use **`typename`** keyword extensively in their implementations

This Week’s Assignment

- Get your source code into a repository
  - Use Subversion – provided on CS cluster
  - Use the CS cluster since it’s backed up
  - Can access remotely, if desired
- Complete basic functionality of the ray tracer
  - Implement last major function on scene object
    - “Trace a ray, and return the color.”
  - Write code for scanning the scene and storing the results in a simple image file format
  - Should be able to render an image this week!