Scientific Computing: Modules, make & git

Erik Spence

SciNet HPC Consortium

14 January 2014

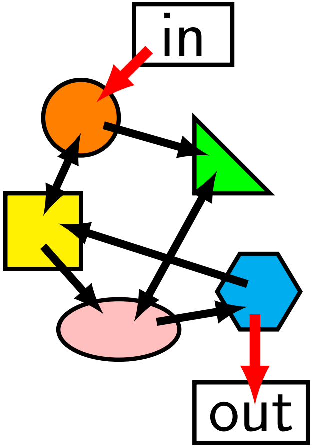
Today's class

Today we will discuss the following topics:

- Modules. How to make them and use them.
- The 'make' command.
- Version control using git.

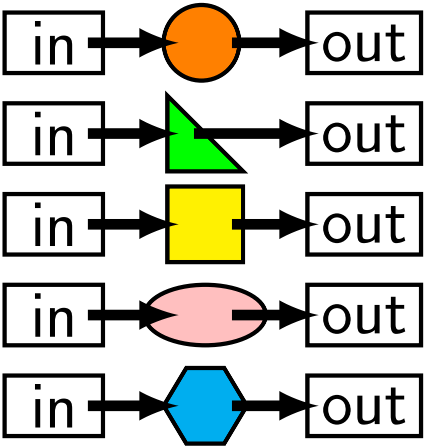
Integrated testing

- As you develop a large program, with many interacting parts, bugs will develop.
- It will be difficult to determine the source of the problem, if testing is only performed on the integrated whole.



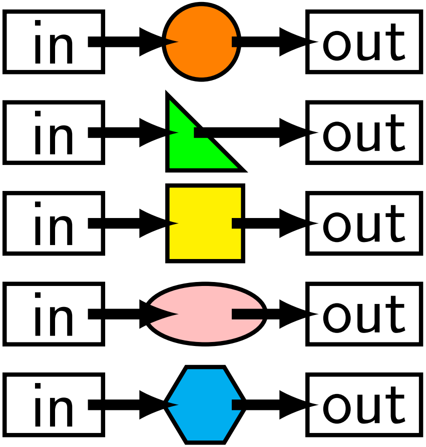
Unit testing

- Test major pieces of the code individually.
- This usually involves creating specialized pieces of software that will interface with the pieces of code being tested.
- Test against easy (or exact) solutions, typical solutions, edge cases, special cases.
- Enormously speeds up and simplifies the detection of bugs.



Testing and Modularity

- Modular software is needed for unit testing. This means creating functions that do one or two specific things, and does them well.
- Break your code up into separate units.
- Also answers the question: "How much should be in a module?" and "What are good independent tests?"
- Define a module to be a testable unit of functionality.



A simple example

Imagine we're writing a matrix in binary and in ascii:

```
// MyArrays.cpp

void toBin(const char *n, double **x, int r, int c) {
  // A bunch of commands.
}

void toAsc(const char *n, double **x, int r, int c) {
  // A bunch of commands.
}

int main() {
  // ...
  toBin("data.bin", data, nrows, ncols);
  // ...
  toAsc("data.txt", data, nrows, ncols);
  // ...
}
```

A simple example, continued

```
// Prototypes.
void toBin(const char *n, double **x, int r, int c);
void toAsc(const char *n, double **x, int r, int c);

int main() {
  // ...
  toBin("data.bin", data, nrows, ncols);
  // ...
  toAsc("data.txt", data, nrows, ncols);
  // ...
}

void toBin(const char *n, double **x, int r, int c) {
  // A bunch of commands.
}

void toAsc(const char *n, double **x, int r, int c) {
  // A bunch of commands.
}
```

Creating modules

The prototypes for the functions are placed in their own 'header' file. The source code for the functions can then be put into its own file.

```
// outputarray.h
void toBin(const char *n, double **x, int r, int c);
void toAsc(const char *n, double **x, int r, int c);
```

```
// MyArrays.cpp
#include "outputarray.h"

int main() {
  // // ....
  toBin("data.bin", data, nrows, ncols);
  // // ....
  toAsc("data.txt", data, nrows, ncols);
  // // ....
}
```

Interface versus implementation

- The implementation - the actual code - goes in the .cpp or 'source' files.
- The interface - what the calling code needs to know - goes in the .h or 'header' files.
- This distinction is crucial for writing modular code.

```
// outputarray.h
void toBin(const char *n, double **x, int r, int c);
void toAsc(const char *n, double **x, int r, int c);
```

Interface versus implementation

- When MyArrays.cpp is being compiled into a .o file it needs to know that there exists out there somewhere functions of the form

```
void toBin(const char *n, double **x, int r, int c);
void toAsc(const char *n, double **x, int r, int c);
```

- This allows the compiler to check the number and type of arguments and the return type (also called the interface).
- It does not need to know the details of the implementation (the source code of the routine).
- Neither does the programmer of MyArrays.cpp (nor does the programmer, in general, want to know the details of the implementation).

Guards against multiple inclusion

- Header files can include other header files.
- It can be hard to figure out what header files are already included in the program.
- Including a header file twice usually leads to doubly-defined entities, which leads to a compiler error.
- The usual solution is to add a 'preprocessor guard' to every header file:

```
// outputarray.h
#ifndef OUTPUTARRAY_H
#define OUTPUTARRAY_H
void toBin(const char *n, double **x, int r, int c);
void toAsc(const char *n, double **x, int r, int c);
#endif
```

Compiling versus linking

```
ejspence@mycomp ~> g++ -Wall -O3 MyArrays.cpp -c -o MyArrays.o
```

- MyArrays.o cannot be executed the way it's been written, since it is missing the routines for toBin and toAsc.
- After MyArrays.o is generated, it must be linked to the relevant .o files (libraries) to that a working executable can be made.
- If you leave out one of the needed .o files you will get a fatal linking error: 'symbol not found'.
- If the files to be compiled and linked are outputarray.cpp, outputarray.h and MyArrays.cpp, then the full compiling commands are:

```
ejspence@mycomp ~> g++ -Wall -O3 outputarray.cpp -c -o outputarray.o
ejspence@mycomp ~> g++ -Wall -O3 MyArrays.cpp -c -o MyArrays.o
ejspence@mycomp ~> g++ outputarray.o MyArrays.o -o MyArrays
```

What goes into the header file?

- At the very least, the function prototypes.
- There may also be constants that the calling function and the routine need to agree on (error codes, for example) or definitions of data structures, classes, etc.
- Often a description of the module and its functions in the comments.
- Usually there is one .cpp/.h pair per module, often more than one routine.
- Not necessarily every function prototype is in the header file, just the public ones. Routines internal to the module are not in the public .h file.

What goes into the source file?

- Everything which is defined in the .h file which requires code that is not in the .h file.
- Internal routines which are used by the routines prototyped in the .h file.
- To ensure consistency, include the corresponding .h file at the top of the file.
- Everything that needs to be compiled and linked to code that uses the .h file.

Why does modularity matter?

- Scientific software can be large, complex and subtle.
- If each section uses the internal details of other sections you must understand the entire code at once to understand what the code in a particular section is doing.
- This is why global variables are bad bad bad!
- Interactions grow as (number of lines of code)^2.
- You **must** enforce boundaries between sections of code so that you have self-contained modules of functionality.
- Each section can then be tested individually, which is significantly easier.

But it's more work up-front

- Think about the blocks of functionality that you are going to need.
- How are the routines within these blocks going to be used?
- Think about all the things that you might want to use these routines for; only then design the interface.
- The interfaces to your routines may change a bit in the early stages of your code development, but if it changes alot you should stop and rethink things – you're not using the functionality the way you expected to.
- Like documentation, thinking about the overall design, enforcing boundaries between modules, and testing, is more work up-front but results in higher productivity in the long run.

make

The typical compilation of a large program is a two-step process:

1. First individually compile all .cpp files, including the necessary .h (header) files, to generate .o (library) files.
2. Link all of the .o files together, including external .so and .a (shared-object and static library files), to generate an executable.

However, it can get complicated and redundant:

- you need to keep track of what depends upon what.
- you need to retype in the entire compilation command every time you need to recompile.
- It's easy to forget all of your compiler flags from one day to the next, as well as the location of external libraries.

It's better to keep all of this information contained in a single file. This is where the 'make' program enters the picture.

make, continued

make is a program that is used to build programs from multiple .cpp, .h, .o, and other files.

- make is a very general framework that is used to compile code, of any type.
- make takes a 'Makefile' as its input, which specifies what to do, and how.
- The Makefile contains variables, rules and dependencies.
- The Makefile specifies executables, compiler flags, library locations, ...
- Use make!

Compiling with make

How does make work?

- A makefile 'rule' is a word followed by a colon.
- By default make will execute the first rule it encounters.
- After the colon are the dependencies of the rule.
- When make hits a dependency it goes and looks for it.
- When it runs out of rules for the dependencies, it checks the timestamps; if the dependency is newer than the rule the command is executed.

```
# This file is called Makefile

# Define the compiler to use.
CPP = g++

# Compiler and linker flags.
CPPFLAGS = -I${GSLINC} -O2
LDFLAGS = -L${GSLINC}
LDLIBS = -lgsl -lgslcblas

all: myprog

myprog: myprog.o
  ${CPP} -o myprog myprog.o \
  ${LDLIBS}

myprog.o: myprog.cpp
  ${CPP} ${CPPFLAGS} -o myprog.o \
  myprog.cpp
```

Compiling multiple source files with make

How does make work?

- make will only recompile those dependencies that have source files that are newer then the library, thus only the code you are working on is modified.
- The most annoying part of make: the indentation of the command after the rule is actually a 'tab', and it must be a tab.
- The \ symbol indicates a line-continuation.

```
# This file is called Makefile
CPP = g++
CPPFLAGS = -I${GSLINC} -O2
LDFLAGS = -L${GSLINC}
LDLIBS = -lgsl -lgslcblas

all: MyArray

MyArray: MyArray.o outputarray.o
	${CPP} -o MyArray MyArray.o \
	outputarray.o ${LDLIBS}

MyArray.o: MyArray.cpp outputarray.h
	${CPP} ${CPPFLAGS} -o MyArray.o \
	MyArray.cpp

outputarray.o: outputarray.cpp
	${CPP} ${CPPFLAGS} \
	-o outputarray.o outputarray.cpp
```

Put a 'clean' rule in your Makefile

```
CPP = g++
CPPFLAGS = -I${GSLINC} -O2
LDFLAGS = -L${GSLINC}
LDLIBS = -lgsl -lgslcblas

MyArray: MyArray.o outputarray.o
  ${CPP} -o MyArray MyArray.o outputarray.o ${LDLIBS}

MyArray.o: MyArray.cpp outputarray.h
  ${CPP} ${CPPFLAGS} -o MyArray.o MyArray.cpp

outputarray.o: outputarray.cpp
  ${CPP} ${CPPFLAGS} -o outputarray.o outputarray.cpp

clean:
  rm -f MyArray.o outputarray.o MyArray
```

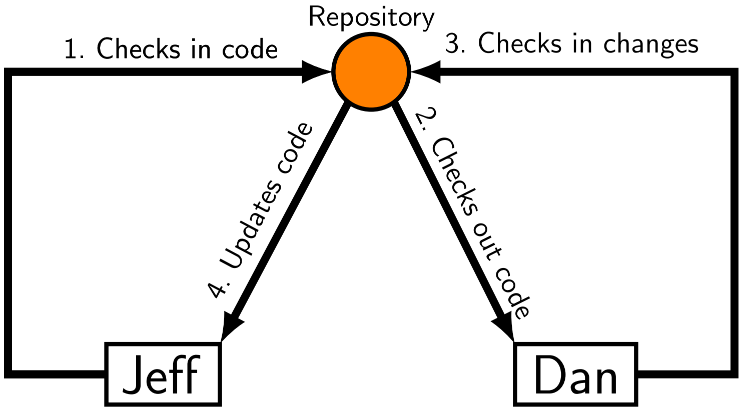
```
ejspence@mycomp> make clean
ejspence@mycomp> make
```

Version control

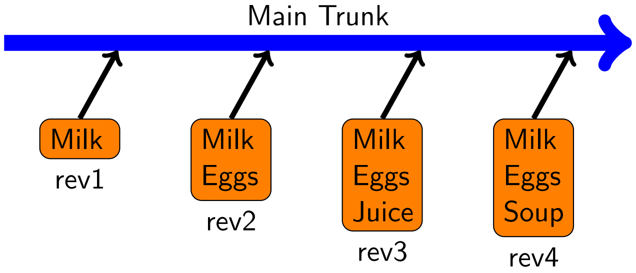
We use version control to save ourselves from ourselves. More specifically:

- Version control is a tool for managing changes in a set of files.
- It is used to figure out who broke what, where and when.
- It essentially takes a snapshot of the files (code) at a given moment in time.
- Why do it?
  - Makes collaborating on code easier/possible.
  - Helps you stay organized.
  - Allows you to track changes in the code.
  - Allows reproducibility in the code.

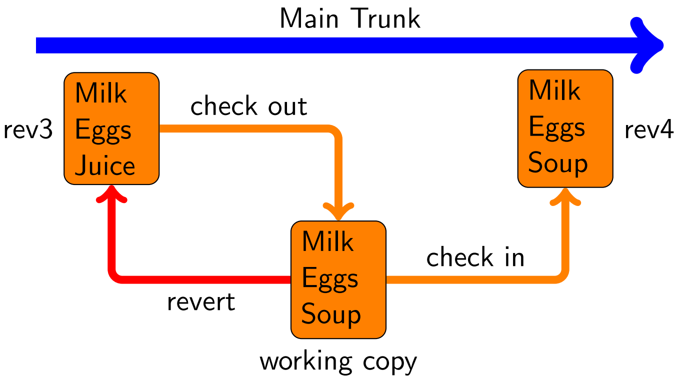
How does version control work?



Basic Checkins



Checkout and edit



Version control: git

There are many types and approaches to version control. Here we will introduce one implementation: git.

There are four main things you need to know how to do to get started with git:

- Initialize a git repository.
- Commit files to the repository.
- Delete files from the repository.
- Where to find more information.

Version control: setup a repository

The first thing to do is set up a repository for your code.

```
ejspence@mycomp ~> cd code
ejspence@mycomp code> git init
Initialized empty Git repository in /home/s/scinet/ejspence/code/.git/
ejspence@mycomp code>
```

This creates a .git directory, in the code directory, which contains the repository information.

```
ejspence@mycomp code> ls -a
.     ..     .git
ejspence@mycomp code>
```

Version control: adding repository files

We now need to add files to the repository. First you must add the files to the 'staging' area, then you commit.

```
ejspence@mycomp code> echo "some data" > temp.txt
ejspence@mycomp code> cp temp.txt temp2.txt
ejspence@mycomp code> ls
temp2.txt     temp.txt
ejspence@mycomp code> git add .  # include all files in the commit.
ejspence@mycomp code> git commit -m "First commit for my repository."
[master (root-commit) f60c07d] First commit for my repository.
2 files changed, 2 insertions(+), 0 deletions(-)
create mode 100644 temp.txt
create mode 100644 temp2.txt
ejspence@mycomp code>
```

Unfortunately, you must always 'stage' the files before commiting them.

Version control: removing repository files

Let's look at what we've done so far.

```
ejspence@mycomp code> git log
commit f60c07da5e36c9dcd55e3e51323391e550c42920
Author: Erik Spence <ejspence@scinet.utoronto.ca>
Date: Wed Jan 8 14:34:31 2014 -0500

First commit for my repository.
```

But suppose you want to delete a file?

```
ejspence@mycomp code> git rm temp2.txt
rm 'temp2.txt'
```

```
ejspence@mycomp code> git add .
```

```
ejspence@mycomp code> git commmit -m "Remove temp2.txt."
[master 95c1ef3] Remove temp2.txt
1 files changed, 0 insertions(+), 1 deletions(-)
delete mode 100644 temp2.txt
```

Version control: more information

There are many other things that can be done with git:

- Review differences between files in different commits.
- Go back to a previous version of the code.
- Branch the code to add new and wonderful features.
- Reconcile different branches of the code.

For a very extensive tutorial, go here:
`http://www.vogella.com/tutorials/Git/article.html`