C/C++ Programming
Session 7

Dr Jim Lupo
LSU/CCT Computational Enablement
jalupo@cct.lsu.edu
### Concept Review

<table>
<thead>
<tr>
<th>int main ( ... ) { ... };</th>
<th>Float precision</th>
<th>if / else if / else</th>
</tr>
</thead>
<tbody>
<tr>
<td>statement;</td>
<td>Promotion</td>
<td>switch</td>
</tr>
<tr>
<td>{ statement block; ... };</td>
<td>Machine $\epsilon$</td>
<td>break / continue / goto</td>
</tr>
<tr>
<td>Integer types</td>
<td>Operators</td>
<td>nesting statements</td>
</tr>
<tr>
<td>Float types</td>
<td>Bool types</td>
<td>data structs</td>
</tr>
<tr>
<td>String constants</td>
<td>Association</td>
<td>pointers</td>
</tr>
<tr>
<td>Variables</td>
<td>Precedence</td>
<td>functions</td>
</tr>
<tr>
<td>cin / cout</td>
<td>for Loop</td>
<td>call sequences</td>
</tr>
<tr>
<td>$&lt;&lt;$ and $&gt;&gt;$</td>
<td>while Loop</td>
<td>do Loop</td>
</tr>
<tr>
<td>Storage in memory</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Anatomy of a Problem

The Laplace Heat Equation lets you determine the steady-state temperature at each point along a bar given the fixed temperatures at each end:

0°F _____________________________ 100°F

Formally, the temperatures are given by the solution of an elliptic partial differential equation:

$$\Delta T = \frac{\partial^2 T}{\partial x^2} = 0$$
Estimating a Solution

A numerical solution is no more than an estimated guess, within some error bounds, of what the temperatures are along the bar. With one point, say the mid-point, the intuitive answer is 50°F

0°F  50°F  100°F

This clearly depends on the temperatures being fixed and no cooling, but those are exactly the conditions the equation requires.
Adding More Points

Dividing in half again would allow 3 points to be estimated:

\[ T_i = 0.5 \times (T_{i-1} + T_{i+1}) \]
General Solution Method

If we always subdivided by 2, we could program a solution, which you might try on your own. But what if the mid-point was not included (i.e. divide into 5 parts)? Now you must weight the temperatures by how far along the bar the points are. It is easier to iterate to a solution. The steps would be:

1. Uniformly divide the bar into segments.
2. Set an initial temperature at points along the bar.
3. Apply the equation at each point.
4. See how much the temperature changed.
5. If greater than desired error, go to 3.
6. Report the temperature at each point.
### Solution Pictogram

**Physical:**

<table>
<thead>
<tr>
<th>Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0°F</td>
</tr>
<tr>
<td>100°F</td>
</tr>
</tbody>
</table>

**Numerical:**

<table>
<thead>
<tr>
<th>Step</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12.5</td>
</tr>
</tbody>
</table>

**Step 371**

<table>
<thead>
<tr>
<th>4.805</th>
<th>14.43</th>
<th>24.12</th>
<th>33.89</th>
<th>43.77</th>
<th>53.77</th>
<th>63.89</th>
<th>74.12</th>
<th>84.44</th>
<th>94.81</th>
<th>100</th>
</tr>
</thead>
</table>

The pictogram shows the temperature changes over steps, with a transition from 0°F to 100°F.
Sample Program

There is a working program to solve this 1-D problem on tezpur:

/home/jalupo/REU_C_Course/S07/laplace-1D.c++
/home/jalupo/REU_C_Course/S07/laplace-1D.c

Login or download a copy to follow along.
Implementation Decisions

In no particular order:

1. How to handle temperature data?
2. Method used to initialize data?
3. Method to set boundary conditions?
4. What user input?
5. Allow output control?
6. Overall code structure?
Code Structure

A quick read through the program shows how 3 of the questions were answered:

- 2, 3, 6 – main and 3 supporting functions.

Good practices to separate overall control of the program from the worker parts.

One modification to try is move input processing to a subroutine.
User Input

Choice of data type related to freedom of choice on number of points. The user may input several control values:

- nmesh ... Number of mesh points across line.
- niter ... Number of iterations to allow.
- iprint .. Iterations between printed output.
- relerr .. The max convergence relative error desired.

A good-enough solution is reached if:

\[ \text{fabs(told-t)/t < relerr} \]
Command Line Arguments

```c
int main( int argc, char *argv[] )
```

The arguments passed to main are defined by the compiler, but nearly always include:

- `argc` .. Number of command line arguments present.
- `argv` .. Array of char arrays holding each argument.

`argv[0]` holds the name of the command executed, so `argc` is never less than 1.
CMATH Functions

The **cmath** header file (or **stdlib.h**) provides a variety of math functions. The three I used are:

- **atoi()**.. converts a string of digits to an int value.
- **atof()**.. converts a string to a double value.
- **fabs()**.. returns the absolute value.

A handy resource describing functions in **cmath** and other available library functions:

http://www.cplusplus.com/reference/clibrary/cmath/

Just “Google” the include file name, and you'll get something useful.
Primary Data

The choice for Question 1 is the use of two arrays, one for the new, or updated, temperatures ($t$), and one for the current, or “old”, temperatures ($t_{old}$).

I've choosen to use double precision. Why?
Allocating Memory

Since I'm letting the user decide how to subdivide the rod, I've elected to use memory allocation to create storage when needed. Since array names are pointers, this isn't too confusing.

t = (double *) calloc( nc+2, sizeof(double) );

Introduces the functions `calloc` and `sizeof`, plus concept of a `cast`, or forced promotion. Returns NULL if unsuccessful (i.e. out of memory):

```c
long *x;
x = (long *) calloc( 1, sizeof(long) );
if ( x == NULL ) { . . . ERROR . . . }
```
Output Control

Iteration is a common solution requirement. Since I don't know in advance how many iteration steps there will be, a standard trick is to get output only every N'th step.

```c++
if ( iprint != 0 ) {
    if ( (iter % iprint) == 0 ) {
        cout << "Iteration: " << iter << "; Max error: " << maxerr << endl;
    }
}
```
Initialize Function

1. int initialize( double *array, int nc )
2. {
3. //________________________________________________________________________
4. //
5. // Make sure the mesh is initialized to 0.
6. //
7. // array .. 1-D array holding mesh data.
8. //
9. //________________________________________________________________________
10. for ( i = 1; i <= nc; i++ ) {
11. array[i] = 0.0;
12. }
13. }
14. // Don't really have any errors to check for.
15. return ( 0 );
16. }
17. // End of int initialize()
BC Function

```c
1. int set_bcs( double *array, int nc )
2. {
3.    /////////////////////////////////////////////////////////////////////////////
4.    //
5.    //  Routine which sets up the fixed boundary conditions.
6.    //
7.    //    array .. 1-D array holding mesh data.
8.    //
9.    /////////////////////////////////////////////////////////////////////////////
10.   array[0] = 0.0;
11.   array[nc+1] = 100.0;
12.  
13.   // Don't really have any errors to check for.
14.  
15.   return ( 0 );
16. }
17. }
18. // End of int set_bcs()
```
for ( iter = 0; iter < niter; iter++ ) {
  for ( i = 1; i <= nc; i++ ) {
    t[i] = 0.5 * ( told[i+1] + told[i-1] );
  }
}

The primary processing loop runs over the allowed number of iterations.

The solver just runs over the data, computing new values.

Rest of primary loop does various checks and output.
maxerr = 0.0;
error = 0.0;

for ( i = 1; i <= nc; i++) {
    if ( t[i] != 0.0 ) {
        // Compute the error.
        error = fabs(told[i]-t[i])/t[i];
    }
    if ( maxerr < error ) {
        // Keep tabs on the maximum error seen.
        maxerr = error;
    }
    told[i] = t[i];
}
Solution Flag

// Now see if the maximum error seen is less than
// the desired error.

solved = maxerr < relerr;

At the very end of the loop, we make use of the value.

// We're done if solved, or run out of iterations.

if ( solved ) break;
Not So Obvious Points

- Avoid mixing tight calculation loops and logic
- Style element – name end of routine.
- Style element – `main` before support functions.
Modifications

1) Modify to use only automatic variables.
2) Rework to use the “intuitive” solution method.
3) Add an output subroutine to produce “pretty” output.
4) Break single source file into multiple files.
Protect Yourself

The most dangerous person a programmer must deal with?

Themself!

The reasons are legion:

• Edits and changes that should just work, but don't.
• No backups.
• Stray "rm -rf *" commands.
• Unthinking slip: `icc -o foo.C foo.C`
• No history of what was done if redesign is needed.
Revision Control Systems

A revision control system manages source code in a way that allows changes to be tracked and previous versions of code recovered.

There are many examples: Mercury (hg), Subversion (svn), CVS (cvs), git, rcs, sccs, update.

The two that are currently popular are:
- git – Very good for highly distributed projects.*
- svn – Most popular for small group and individual projects.**

* [http://git-scm.com/documentation](http://git-scm.com/documentation)
** [http://subversion.apache.org/docs/](http://subversion.apache.org/docs/)
Terminology and Process

The *repository* stores the data and change information.

The *working copy* is the data undergoing modification and use.
The Repository

A subdirectory created with SVN tools that contains the files needed to make the magic work. These are portable – you can copy the entire repository to a new machine and it will continue working.

A single repository can contain multiple projects. Each project should be related to a single application or purpose.

Any subdirectory tree used to organize the project can be maintained by the repository!

The repository can be reorganized by reorganizing the work copy.
Preliminaries

We're going to be working with the Laplace program, so before any modifications, it should be placed in a repository for safe keep.

Creation of a repository is handled by the command: `svnadmin`

```
$ svnadmin create /home/jalupo/Dev/Repo
```

“Repo” is the name of the repository. On Tezpur it must reside in `/home` because of file system capability requirements.
Repo Contents

Repo:
conf/  dav/  db/  format  hooks/  locks/  README.txt

Repo/conf:
svnserve.conf

Repo/dav:

Repo/db:
changes  __db.002  __db.005  log.0000000001  revisions  uuids
copies   __db.003  DB_CONFIG  nodes  strings
__db.001  __db.004  fs-type  representations  transactions

Repo/hooks:
post-commit.tmpl  pre-commit.tmpl  start-commit.tmpl
post-revprop-change.tmpl  pre-revprop-change.tmpl

Repo/locks:
db.lock  db-logs.lock
Getting Help

If you need a reminder of how things work, you can request help:

```bash
$ svnadmin help
```

General usage: `svnadmin SUBCOMMAND REPOS_PATH [ARGS & OPTIONS ...]

Type "svnadmin help <subcommand>" for help on a specific subcommand.

Available subcommands:
- create
- deltify
- dump
- help (?, h)
- hotcopy
- list-dblogs
- list-unused-dblogs
- load
- lstxns
- recover
- rmtxns
- setlog
- verify

```bash
$ svnadmin help create
```
Create A Project

Creating a project creates a subdirectory in the repository with the same name. Let's call it “Laplace”:

```
$ export REPO=file:///home/jalupo/Dev/Repo
$ svn mkdir -m "Project directory for Laplace" $REPO/Laplace
$ svn list $REPO
Laplace/
$ svn list $REPO/Laplace
$
```

Apparently there is nothing in it just yet, but let's talk about what was done.
Repository URL

$ export REPO=file:///home/jalupo/Dev/Repo

This line defined a shell variable to make it easy to reference the repository URL (universal resource locator). It can take several forms depending on the different services which can support SVN:

file://  svn://

All repositories are created using file://. The other forms require someone to set up various services (web, ftp) so others can access the repository.
Checkout (co) the Project

Create a *working copy* by checking out the desired project:

```
$ svn co $REPO/Laplace
```

You should now have a new subdirectory named `Laplace`, which holds 1 hidden subdirectory named `.svn`. This subdirectory holds data relating the working copy with the repository copy. Changes can now be tracked.
Getting Help

If you forget which command to use, or the command line args, help is available just as with `svnadmin`:

```bash
$ svn help
usage: svn <subcommand> [options] [args]
Type "svn help <subcommand>" for help on a specific subcommand.

Most subcommands take file and/or directory arguments, recursing on the directories. If no arguments are supplied to such a command, it will recurse on the current directory (inclusive) by default.

Available subcommands:
    add
    blame (praise, annotate, ann)
    cat
    checkout (co)
    cleanup
    . . . deleted for brevity's sake ...
```
Add the First File

Copy or create a file in the Laplace directory to start things off:

```
$ cd Laplace
$ cp /work/jalupo/Dev/C_Course/S07/laplace-1D.c++ .
```

Its just another copy until we **add** and **commit** it to the repository:

```
$ svn add laplace-1D.c++
$ svn commit -m "Initial version"
```

**add** signals our intent to include the file in the repository, but **commit** makes it happen. **-m** provides a log message.
Sequence of Events

1. The capital A indicates the file has been marked for add.
2. Note the “Transmitting file data” message.

This should make it obvious we're dealing with a two-step process.

$ cp ./S07/ laplace-1D.c++ .
$ svn add laplace-1D.c++
A laplace-1D.c++
$ svn commit -m "Initial commit."
Adding laplace-1D.c++
Transmitting file data.
Committed revision 2.
Revision Numbers

The revision number applies to the entire project. It is not the revision sequence of a individual file. A project might be at revision 95, but some files may never have changed, and others may have changed only a few times. This command:

```
$ svn co -r 42 Laplace
```

would check out the file versions that were current at revision 42. If we make any major development errors, we can recover a working copy and start comparing with the broken one to see where things might have gone astray.
File Status

Let's make an innocuous change, say change `maxiter` from 1000000 to 2000000, and see what `svn` can do.

We can check for changes (i.e. the `status` of the working copy):

```
$ svn status
M   laplace-1D.c++
$ 
```

Well, we knew the file had changed, but `svn` knows too and shows the file state is `M` (modified). You could also do:

```
$ svn status laplace-1D.c++
```
File Differences

We can quickly see what has changed between the modified file and the latest version in the repository:

```
$ svn diff laplace-1D.c++
Index: laplace-1D.c++
=====================================================================
--- laplace-1D.c++ (revision 2)
+++ laplace-1D.c++ (working copy)
@@ -13,7 +13,7 @@

double relerr;

-const int maxiter = 1000000;
+const int maxiter = 2000000;

int laplace( void );
int set_bcs( double *, int);
```
What's it All Mean?

The output follows the unified diff format.

```diff
--- laplace-1D.c++ (revision 2)
 +++ laplace-1D.c++ (working copy)
 @@ -13,7 +13,7 @@

 The change context starts at line 13 and is 7 lines long in rev 2, and
 starts at line 13 and is 7 lines long in the working copy. This helps
determine where in the file the change occurs.

-old
+new

-const int maxiter = 1000000;
+const int maxiter = 2000000;
```

The actual difference is just one line.
Commit the Changes

$ svn commit -m "Increased maxiter."
Sending     laplace-1D.C
Transmitting file data .
Committed revision 3.
$

The commit was successful, and logs the nature of the change. Indeed, if we ever forget what we did, we can view the log!
$ svn log laplace-1D.C

r3 | jalupo | 2012-07-05 14:43:03 -0500 (Thu, 05 Jul 2012) | 1 line

Increased maxiter.

r2 | jalupo | 2012-07-05 13:50:20 -0500 (Thu, 05 Jul 2012) | 1 line

Initial commit.
Best Practice Can Save You

Tracking changes is cheap, so *commit early and often*!

If you mess up badly, or blow something away, you can recover:

```
$ svn update
```

will restore any missing file to the current revision in the working copy. If need be, you could check out an even earlier version.

If you are doing collaborative development, update will flag conflicting changes so you can work to resolve them.
Rearranging is Cheap

Because change tracking is cheap, it is easy to rearrange the source tree to match growth needs. The hardest thing to do is remember to use the **svn** commands, and not the normal OS commands:

```
svn rename
svn mv
svn copy
svn rm
```

A delete or `rm` doesn't actually delete anything. It simply marks the files as not present in the current revision! As long as you don't delete the repository, you can recover anything that was in it.
Comments?

Questions?