C/C++ Programming
Session 8

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 $\mathbb{E}$</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>program walk-through</td>
</tr>
<tr>
<td>Storage in memory</td>
<td>do Loop</td>
<td>Revision Control</td>
</tr>
</tbody>
</table>
Current State

We ended last time with a repository set up for the Laplace program. Let's break the single source file into 4 files:

main.c  laplace.c  initialize.c  set_bcs.c

Let's start by compiling each file, one at a time, and capture the output for analysis (will use a bash shell “for” loop):

$ for f in main laplace initialize set_bcs; do
  $ icc -c -diag-file $f.c
  $ done

The -diag-file switch places error messages in files named $f.diag, where the $f is replace with one of the names listed.
Results

```
$ for f in main laplace initialize set_bcs; do
  > icc -c -diag-file $f.c
  > done

compilation aborted for laplace.c (code 2)
compilation aborted for initialize.c (code 2)
$ ls
initialize.c  laplace.c  main.c  main.o  set_bcs.diag
initialize.diag laplace.diag main.diag set_bcs.c  set_bcs.o
[jalupo@tezpur1 Laplace]$
```

So it looks like two of the files compiled correctly, and two generated error messages. Let's see how bad things are:

```
$ wc *.diag
   4  18  100 initialize.diag
  16  59  472 laplace.diag
   0   0   0      main.diag
   0   0   0   set_bcs.diag
  20  77  572      total
$
Fixing initialize.C

Let's see what error messages were produced:

$ cat initialize.diag
initialize.c(11): error: identifier "i" is undefined
   for ( i = 1; i <= nc; i++ ) {
     ^
$  

This is an easy one. The compiler is complaining because a variable i was seen, but no matching definition (i.e. no type or memory location is available to it).
Understand the Problem

What happened?

There are two ways to fix this problem:

1. Make it a global variable using an `extern` declaration.
2. Define an automatic variable within the function.
Using extern

The required modification is pretty simple:

```c
extern int i;

int initialize( double *t, int nc )
{
```

This is expedient, and maintains the same semantics as the original program.

Clearly, not good for reuse of this function in some other programs! It requires i to be defined somewhere!
New Automatic Variable

This approach is equally simple:

```c
int initialize( double *t, int nc )
{
    int i;
}
```

Since the original program never depended on using i to pass data from 1 function to another, This is likely the better choice. But, I'll stay with the first approach for now.
Fixing laplace.c

There are 4 error messages:

```
laplace.c(51): error: identifier "nc" is undefined
    t = (double *) calloc( nc+2, sizeof(double) ); ^

laplace.c(66): error: identifier "niter" is undefined
    for ( iter = 0; iter < niter; iter++ ); ^

laplace.c(99): error: identifier "relerr" is undefined
    solved = maxerr < relerr; ^

laplace.c(103): error: identifier "iprintf" is undefined
    if ( iprintf != 0 ) { ^
```
How To Fix?

Looks like the error involving `nc` is another scope problem. That we can fix like the last one. But is there a difference to be worried about – `extern` or new automatic?

How do we get the definition of `calloc`?

How should we declare the two functions?
First Set of Fixes

```
#include <stdio.h>
#include "initialize.h"
#include "set_bcs.h"

extern int nc;

int laplace( void )
{

initialize.h contains only:
int initialize( double [], int );

set_bcs.h contains only:
int set_bcs( double [], int );
```
Some Other Errors

laplace.c(54): error: identifier "niter" is undefined
   for ( iter = 0; iter < niter; iter++) {
       ^

laplace.c(67): error: identifier "abs" is undefined
   error = abs(told[i]-t[i])/t[i];
       ^

laplace.c(70): error: identifier "relerr" is undefined
   error = 2.0 * relerr;
       ^

laplace.c(85): error: identifier "iprint" is undefined
   if ( iprint != 0 ) {
       ^
The Final `laplace.c` Fix

```c
#include <stdio.h>
#include "initialize.h"
#include "set_bcs.h"

extern int nc;
extern int niter;
extern int iprint;
extern double relerr;

int laplace( void )
{
```
Adjust main.c And Go

We can remove the include of the `<math.h>`.

```
$ rm *.diag
$ for f in main laplace initialize set_bcs; do
> icpc -c -diag-file $f.c++
> done
$ wc *.diag
 0 0 0 initialize.diag
 0 0 0 laplace.diag
 0 0 0 main.diag
 0 0 0 set_bcs.diag
 0 0 0 total
$

Look Ma! No errors!
Finish the Build

This command is all that is needed:

```
$ icc -o laplace *.o
```

It is commonly called the *link* step, as object modules are linked together to form the executable.
And Test!

$ ./laplace 100 10000 1000 0.0001
Iteration: 0; Max error: 1.000000e+00
Iteration: 1000; Max error: 9.152184e-03
Iteration: 2000; Max error: 2.044386e-03
Iteration: 3000; Max error: 7.984576e-04
Iteration: 4000; Max error: 3.870803e-04
Iteration: 5000; Max error: 2.086444e-04
Iteration: 6000; Max error: 1.190356e-04
Stopped after 6326 iterations.
Max error: 9.989851e-05
Temps:
1   :  8.973750e-01
2   :  1.794929e+00
3   :  2.692484e+00
4   :  3.590575e+00
5   :  4.488667e+00
96  :  9.458781e+01
97  :  9.566953e+01
98  :  9.675197e+01
99  :  9.783441e+01
100 :  9.891720e+01
Solution did converge.

It still gives the same answer as documented in the source file!
Save That Work!

Time to give **svn** a workout.

Check file status:

```
$ svn status
?   laplace
?   initialize.h
?   set_bcs.h
M   initialize.c
M   laplace.c
M   main.c
```

Add the header files. Don't add laplace. It can be rebuilt.

```
$ svn add *.h
A   initialize.h
A   set_bcs.h
```
Commit For Posterity Sake

$ svn commit -m "Working multifile version."
Sending initialize.c
Adding initialize.h
Sending laplace.c
Sending main.c
Adding set_bcs.h
Transmitting file data ..... 
Committed revision 5.
$

This leaves us at revision 5. All the changes are captured, and we could “roll back” to previous versions if we discover any major problems as we go forward.
Automating The Build

We did have some confusion over the use of `icc` versus `icpc`, so maybe we should do something to avoid it in the future?

For that we will use `make`, a special kind of scripting language that keeps track of `dependencies` and can be used to record hard-won knowledge.

By default, `make` will first search for a file named `makefile`, or `Makefile`, in that order, to get its instructions. *(Note: this is actually version dependent behavior – all versions let you specify a file name with the `-f` switch).*
Dependencies

What is a dependency? It is a condition that must be meet before some action can be correctly taken.

Think of how we are compiling the program:

$ icc -c name.c
$ icc -o laplace name.o n2.o n3.o . . .

If we make any editorial changes to the source files, what actions do we need to take to rebuild laplace?
## Steps Based on Changes Made

<table>
<thead>
<tr>
<th>If we change:</th>
<th>We must:</th>
</tr>
</thead>
</table>
| **name.c**    | Produce new **name.o**  
|               | Produce new **laplace** |
| **set_bcs.h** | Identify **name.c** files that depend on it.  
|               | Produce new **name.o** files  
|               | Produce new **laplace** |

(Targets)  (Dependencies)

In the language of **make**, we produce a **target** from **dependencies**.
Break It Down

Target `laplace depends` on 4 object module files.

Each target object module depends on a source file.

Some source files depend on header files, which means some object module files depend on both a header file and a source file.
A First Make File

1. laplace : main.o initialize.o laplace.o set_bcs.o
2. icc -o laplace main.o initialize.o laplace.o set_bcs.o
3.
4. main.o : main.c
5. icc -c main.c
6.
7. initialize.o : initialize.c
8. icc -c initialize.c
9.
10. laplace.o : laplace.c initialize.h set_bcs.h
11. icc -c laplace.c
12.
13. set_bcs.o : set_bcs.c
14. icc -c set_bcs.c

# ALERT – these indents are literal tab (\t) characters, not spaces.

Nothing fancy – exactly what we did manually.
How To Test?

You can exercise `make` by using the `-n` (no action) switch. It will simply report what `make` would try to do when run for real.

Let's clean out the directory of all the `*.o` and laplace file, then see what happens:

```
$ rm *.o laplace
$ make -n
icc -c main.c
icc -c initialize.c
icc -c laplace.c
icc -c set_bcs.c
icc -o laplace main.o initialize.o laplace.o set_bcs.o
$
Run for Real

#1
$ make
  icpc -c main.c++
  icpc -c initialize.c++
  icpc -c laplace.c++
  icpc -c set_bcs.c++
  icpc -o laplace main.o initialize.o laplace.o set_bcs.o
$  
#2
$ make
make: `laplace' is up to date.
$

Note the second time `make' was run, it didn't do anything.
Last Modified DateTime

Make checks the last modified date and time. If a target is older than a dependency, it rebuilds the target. The touch command updates the last modified date/time and provides a handy way of testing:

```
$ touch laplace.c
$ make
icpc -c laplace.c
icpc -o laplace main.o initialize.o laplace.o set_bcs.o
$ touch set_bcs.h
$ make
icpc -c laplace.c
icpc -o laplace main.o initialize.o laplace.o set_bcs.o
$  
```
Add a `clean` Target

Many source distributions include a target that cleans up all the auxiliary files used to construct an executable target (i.e. `.o` files). They might also add a `distclean`, which means leave only the original distribution files behind.

A target does not require a dependency, and may list only actions. Let's add the following lines to the Makefile:

```makefile

clean:
    rm -f *.o

distclean: clean
    rm laplace
```
Test the Changes

$ make -n clean
rm -f *.o
$
$ make -n distclean
rm -f *.o
rm -f laplace
$

This relies on the ability to specify a target as a command line argument to **make**.

(Default is the first target encountered)
Make Macros

Macros serve as a powerful shorthand device to save typing. They are very much related to shell environment variables, in fact and in behavior. Let's fix the compiler usage first:

```
CC = icc

laplace : main.o initialize.o laplace.o set_bcs.o
  ${CC} -o laplace main.o initialize.o laplace.o set_bcs.o

main.o : main.c++
  ${CC} -c main.c++

... lines removed for brevity ...

clean :
  rm -f *.o

distclean : clean
  rm -f laplace
```
Pattern Rules

Formally, the line:

```
target : dependency
```

is called a *rule*. We can write a general compile pattern rule:

```
%.o : %.c
${CC} -c $< -o $@
```

This uses two built-in macros (of many):

```
$<  ..  Name of target
$@  ..  Name of dependency
```
Very Compact Makefile

CC = icc

%.o : %.c
    ${CC} -c $< -o @$

laplace : main.o initialize.o laplace.o set_bcs.o
    ${CC} -o laplace $$

laplace.o : initialize.h set_bcs.h

clean :
    rm -f *.o

distclean : clean
    rm -f laplace

$$ .. All dependencies listed macro.
M⁴ (Much, Much More Make)

More automatic variables available.
Conditional statements.
String functions.
Shell commands.

**make** is a language onto itself.
Take-Aways

Be aware of variable scope.

Check in early and often.

Use make to automate everything.

Allow for build AND cleanup.
Comments?

Questions?