9.   6810 Session 9

a.   Follow-ups to Session 8 and earlier

- **"Visualization of the Pendulum's Dynamics."** The handout with this heading shows the progression of phase-space trajectories and Poincaré sections for a pendulum as the value of the driving force amplitude is varied. The idea of a Poincaré section is to plot a point in phase space once every period of the external force, $2\pi/$(external frequency). The resulting pattern gives information about the periodicity of the signal (or indicates chaos). The green points in the gnuplot plots from Session 8 form a Poincaré section. You will also find the Mathematica version in the `nonlinear.nb` notebook in Session 10. The figure caption to Fig. 3.4 in the handout indicates the key characteristics of each plot.

  Figures 3.8 and 3.9 show another analysis tool: the *power spectrum*. The power spectrum is found by taking a Fourier transform (FFT) of the "signal" (e.g., the angle of the pendulum as a function of time). In Fig. 3.8, we see discrete frequencies, while in Fig. 3.9 we see a continuous distribution, which is an indicator of chaos. In the Session 10 `nonlinear.nb` and `pendulum.nb` Mathematica notebooks, you'll be able to create power spectra.

- **Follow-up to Streams.** In the `filename_test.cpp` code, we created stringstream objects and used them to create filenames to open. In doing so, we had an intermediate step where we defined a string `filename2`:

  ```
  // next, create a string with the stringstream class
  ostringstream filename_stream; // declare a stringstream object

  // you can load the string stream just like output streams
  filename_stream << "test_stream" << i << ".out";
  // use .str() to convert to a string
  string filename2 = filename_stream.str();

  ofstream file2;              // print to file2, which is named by filename2
  file2.open (filename2.c_str()); // use .c_str() to convert to a char *
  ```

  We used the intermediate string `filename2` because we wanted to first convert `filename_stream` to a string, and then convert the string to a C-style string (using `.c_str()`). We can skip the intermediate step and do it all at once with `.str().c_str()`:

  ```
  // next, create a string with the stringstream class
  ostringstream filename_stream; // declare a stringstream object

  // you can load the string stream just like output streams
  filename_stream << "test_stream" << i << ".out";

  ofstream file2;       // print to file2, which is named by filename_stream
  file2.open (filename_stream.str().c_str()); // convert to a char *
  ```

b.   A Bit More on C++ Classes

In Session 9 we'll backtrack to the very first session and reconsider the programs to calculate the area of a circle and the volume of a sphere. The focus of `area.cpp` was on the formula for calculating the area of a circle. This is a simple example of the procedural approach to programming. To move toward object oriented programming, we would focus instead on the circle itself. So we'll construct a Circle *class* that is initiated with a radius. This moves details of the area calculation (like defining π and the implementation of the area formula) out of sight, but also allows us to easily extend the functionality. For example, to calculate the circumference of the circle, we just need to add a method to the class and it is available to all instances of Circle.

In classes, we distinguish between private and public functions and data. The public functions can be called from outside the class while private functions are only available inside the class. Similarly, public variables (data) can be directly changed by outside parts of the program, while private variables can only be changed by using "accessor" functions (see the Session 8 notes and earlier discussions of classes). We'll play in Session 9 with a simple test file called `private_vs_public.cpp` to explore the distinctions.

If we follow Session 1, it is simple to create a Sphere class analogous to the Circle class. Having done so, we observe that they have various features and methods in common. To take advantage of this, we can define a more general class and make Circle and Sphere subclasses, which *inherit* variables and methods. We'll do that later.

c.   Segmentation Faults

[This discussion assumes access to gdb and is based in large part on the article at http://www.cprogramming.com/debugging/segfaults.html.]

One of the most useful functions of a debugger is to track down the source of "Segmentation fault" error messages. In general, a segmentation fault means you have tried to access memory that doesn't belong to your program. This is not allowed, and the segmentation fault or "segfault" is the result. (Note: if you are using Windows but not Cygwin, you may just get a pop-up window saying that the program has encountered an error and must terminate.) One region of memory is called the "stack," which is where local variables are stored. Another region is called the "heap," which is dynamically allocated when your program is running if you use a "new" command in C++.

Here are some sources of segmentation faults (not a complete list!):

1. Accessing an array beyond its bounds. When you declare or allocate an array, it has a certain size. If you try to write beyond it, you will get a segmentation fault error *if* you go outside the memory assigned to your program. In this case, you can use the gdb debugger; it will identify that your program crashes at a line involving an array access and you can print the argument of the array. The tricky case is if you stay within the memory assigned, which means you will not get an error but will unpredictably write over other variables. This is not good!

2. Dereferencing NULL, uninitialized, or deleted pointers. This code illustrates the first one:

   ```
   int *my_ptr = 0;
   *my_ptr = 3;
   ```

   This fails because the pointed is initialized to NULL (which is the same as 0) in the declaration, and then in the next line it is accessed. In gdb, this would be revealed by

   ```
   (gdb) print my_ptr
   $1 = 0x0
   ```

   where memory address 0x0 is, in fact, NULL. To avoid problems with unitialized pointers, you should simply set them to NULL when you declare them (unless you are using them with a "new" statement to allocate memory).

3. Using all of the stack space (such as by a recursive function); this is sometimes reported as a "stack overflow" rather than a segmentation fault.

The use of a debugger such as gdb to track down segmentation faults will be demonstrated in this session. For Linux (only), there is also the very useful Valgrind tool suite, which includes a memory checking tool to detect common memory errors.

d.   Optimization

In an ideal world, optimization of a computer code would be transparent to the user: the compiler would do it for you. In practice, different compilers for the same language on the same machine can provide very different performances. That is why people still pay big bucks for fortran compilers rather than use gfortran (the compiler that is part of the gcc/g++ family). In the Linux C++ world, we have g++ and then commercial compilers; in the latter class is the Intel C++ compiler icpc. It has a strong benchmarking record, particularly on certain platforms. The generic options (i.e., `-O2` or `-O3`) will generally do most of the useful optimizations. But you should be aware that there are many additional optimization options that can improve particular codes, especially on a known architecture.

   In this session, we'll take a look at a simple example of how coding the same operation different ways can make a dramatic difference in the execution time. For example, if we need to calculate the value of x^n where n is an integer, using the function `pow(x,n)` takes much longer than multiplying x together n times. That is because `pow` is a library function valid for any real value of n (i.e., any double). The operations needed in general (e.g., logarithms) take much more time than floating point multiplies and if the compiler doesn't substitute for the general algorithm, there will be a big difference in times. We'll see this in practice.

   Since operations like `exp`, `sin`, and `cos` are also expensive, if they are evaluated repeatedly with the same argument it is often efficient to use a "look-up" table. This is an array that is filled with the needed values once at the beginning of the program, and then just referenced later. An array lookup is much faster, as long as their is memory available for the array.

Optimization Options for g++. If you consult `man g++` you'll find a multitude of options tailoring the optimization of your code with the g++ compiler. The general options `-O0` through `-O3` turn on collections of these options:

`-O0` Do not optimize.

`-O1` These optimizations strive to reduce code size and execution time, using optimizations that do not take a lot of compilation time. It turns on these opimization flags:

```
-fdefer-pop -fmerge-constants -fthread-jumps -floop-optimize
-fif-conversion -fif-conversion2 -fdelayed-branch
-fguess-branch-probability -fcprop-registers
```

`-O2` Do all of the `-O1` optimizations plus many more:

```
-fforce-mem -fopti- mize-sibling-calls -fstrength-reduce
-fcse-follow-jumps -fcse-skip-blocks -frerun-cse-after-loop
-frerun-loop-opt -fgcse -fgcse-lm  -fgcse-sm  -fgcse-las
-fdelete-null-pointer-checks -fexpensive-optimizations -fregmove
-fschedule-insns  -fsched- ule-insns2 -fsched-interblock
-fsched-spec -fcaller-saves -fpeep-hole2 -freorder-blocks
-freorder-functions -fstrict-aliasing -funit-at-a-time
-falign-functions  -falign-jumps -falign-loops -falign-labels
-fcrossjumping
```

`-O3` Do all of the `-O2` optimizations as well as

```
-finline-functions -fweb -frename-registers
```

The man pages describe each of these options, although the explanations are not very clear to the non-expert. The basic message is that a lot of processing is going on behind the scenes to try to make the code run faster. You should also consider the hardware-specific options, such as `-march=`*arch* where *arch* specifies a cpu architecture. See https://wiki.gentoo.org/wiki/GCC_optimization for details.

e.   Profiling

The point of "profiling" is to identify areas of code that use the most overall time. There is no point in optimizing sections that use a small fraction of the total time, particularly if it causes the code to be less clear. For example, if a code spends roughly 90% of its time in one function and 10% in another, making the latter run ten times faster (which is an enormous improvement) will only make the code run 10% faster (e.g., 100 minutes before and 91 minutes after). Focus on the first function!

There is a standard GNU tool for profiling, called `gprof`. We will try a quick demo of what it does. [Unfortunately, `gprof` is not supported on Mac OS X and works erratically on Cygwin. We'll try it out on Linux.]

f.   Bash and Tcsh Shells

The shell is the command-line user interface to the Linux kernel. It starts up when you log into a machine (or bring up a terminal window) and lets you run programs and interact with the computer hardware. You can find out the shell you logged on into by checking the `SHELL` environment variable. Type

```
echo $SHELL
```

at the prompt. The result is most likely to be `/bin/bash` or `/bin/tcsh`. The first one is the "Bourne Again Shell" or bash (which is a descendent of the Bourne shell written by S.R. Bourne) and the second one is is the TC-Shell or tcsh (which started at Carnegie-Mellon University but was then further developed at Ohio State). We had an optional

In Session 5, we (optionally) looked at creating aliases in each of the shells. In Session 9 we'll use some aliases to set up rsync. Later we'll come back and discuss environment variables more so that we can set up our computer to use a different compiler from g++.

g.   Using Rsync for Backups

Here is a quote from the rsync web page (at https://rsync.samba.org/):

> rsync is a file transfer program for Unix systems. rsync uses the 'rsync algorithm' which provides a very fast method for bringing remote files into sync. It does this by sending just the differences in the files across the link, without requiring that both sets of files are present at one of the ends of the link beforehand.
>
> Some features of rsync include
>
> - can update whole directory trees and filesystems
> - optionally preserves symbolic links, hard links, file ownership, permissions, devices and times
> - internal pipelining reduces latency for multiple files
> - can use rsh, ssh or direct sockets as the transport
> - supports anonymous rsync which is ideal for mirroring

There are many options when invoking rsync. In Session 9, two sets of options useful for backing-up and mirroring files to another computer are given (in the form of aliases in a `.bashrc` file). A "mirror" means an exact copy on the destination computer of the files on the source computer (this usually means the files in a given directory and its subdirectories). This means that any files on the destination computer that are *not* on the source computer are deleted. A "back-up" omits this last part, so all of the source files will be reproduced on the destination computer, but there may be additional files.

You will find an explanation of how rsync works (theory and implementation), plus links to the original technical report and Andrew Tridgell's PhD thesis, at
https://rsync.samba.org/how-rsync-works.html

Many helpful resources (such as tutorials and links to helper programs built on rsync) can be found at: https://rsync.samba.org/resources.html

h.   References

[1] R.H. Landau and M.J. Paez, *Computational Physics: Problem Solving with Computers* (Wiley-Interscience, 1997). [See the 6810 info webpage for details on a new version.]

[2] M. Hjorth-Jensen, *Lecture Notes on Computational Physics* (2009). These are notes from a course offered at the University of Oslo. See the 6810 webpage for links to excerpts.

[3] W. Press *et al.*, *Numerical Recipes in C++*, 3rd ed. (Cambridge, 2007). Chapters from the 2nd edition are available online from http://www.nrbook.com/a/. There are also Fortran versions.