C/C++ Programming
Session 9

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 $\mathcal{E}$</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>multi-file</td>
</tr>
<tr>
<td>strings</td>
<td>member functions</td>
<td>svn and make</td>
</tr>
</tbody>
</table>
File I/O

Most real projects involve lots and lots of data, usually contained in files. Learning how to manipulate files is critical to production programming.

Need to come to grips with:

- File types – what sort of data do they contain?
- File access – reading, write, something else?
- File management – testing readability, existence, etc.?
File Types: Text

There are two basic types: text and binary.

Text files – human readable, editable. For instance:

C/C++ source files are text files.

Shell scripts are text files.

stdin, stdout, stderr are text files.

Reading/writing such files is a matter of ingesting or outputting characters.
File Types: Binary

Binary files contain information in raw byte form. For instance:

- Object files and linked executables.
- Many system application commands.
- MySQL databases.
- Crash dumps, restart files, data files, etc.

Any type of data can be contained in a binary file, including text. Just have to know what it is to read it properly.
File Access

You are likely familiar with the concepts of opening and closing a file – done all the time in Windows, with editors, etc.

**open** – alert the operating system you want access to a file.

**close** – alert the operating system you are done with a file.

The O/S performs its “magic” - checks permissions, provides the programming interface to the file system, flushes buffers when done, etc.

The O/S also controls the initial position in the file, and what to do based on whether or not a requested file exists.
C:

```c
#include <stdio.h>

FILE *fh;

fh = fopen(“name”, mode);
fclose( fh);
```

```c
mode: “a” – append
    “r” – read
    “w” – write
    “xb” – add b for binary
    “x+” – add + to read and write

FILE *stderr, *stdout, *stdin are provided.
```

C++:

```cpp
#include <iostream>
#include <fstream>

using namespace std;
ofstream fh

fh.open(“name”, mode);
fh.close();
```

```cpp
mode: ios::app – append
    ios::in – read
    ios::out – write
    ios::binary – for binary

Combine with logical or:
    ios::out | ios::binary
    ios::in | ios::out (r/w)
```
Examples

C:

fh = fopen( "..bashrc", "r" );
fh = fopen( "main.o", "rb" );
fh = fopen( "data.42", "a" )
fh = fopen( "data.100", "wb" )

C++:

fh.open( "..bashrc", ios::in );
fh.open( "main.o", ios::in | ios::binary )
fh.open( "data.42", ios::out | ios::app )
fh.open( "data.100", ios::out | ios::binary )

Rules to keep in mind:

Open for read, and file does not exist or no read permission?  Error!
Open for write, and no write permission?  Error!
Open for write and file exists?  Truncate!
Open for write and file does not exist?  Create!
Simple Reads & Writes

C:

```c
int ans;
char line[100];
fh = fopen( "./bashrc", "r" );
ans = fgetc( fh );
fgets( line, 99, fh );
fclose( fh );
fh = fopen( "foo.dat", "w" );
fprintf( fh, "%s\n", "Hello!" );
```

C++:

```c++
char ans;
string line;
fh.open( "./bashrc", ios::in );
fh >> ans;
getline( fh, line );
fh.close();
fh.open( "foo.dat", ios::out )
fh << "Hello!" << endl;
```

If `<cstdio>` is used in C++, C-style file handling and I/O is supported. In many ways, it is more versatile than using C++ insertion and extraction operators.
The Gory Details of `fprintf`

The prototype is:

```c
int fprintf ( FILE *fh, const char *format, val1, ... , valN )
```

The second argument is a string containing 1 or more format specifiers, each of which has the form:

`%[flag][width][.precision][modifier]specifier`

[] indicates optional elements – better have a cheat sheet to track all the players!

For each specifier, a value must be provided as an added argument.
A Plethora of Write Specification Settings

Flags: -, +, (space), #, 0
Width: an integer value, or * for default based on arg type.
Precision: an integer value, or *
Modifiers: hh, h, (space), l, ll, j, z, t, L
Specifiers: %, c, s, d, i, o, x, X, u, f, F, e, E, a, A, g, G, n, p

These let you position output precisely on a line in a field that might be 0 padded, with or without a leading +, and using decimal, hexadecimal, octal, or engineering notation.

See, for instance: http://en.cppreference.com/w/cpp/io/c/printf
Underlying Rational for width

The ability to specify width of an output field allows creation of column-aligned printout.

Eases the human burden of reading the data.

Eases the programming burden of reading the data back into an application at run time. (The *parsing* problem).
Underlying Rational for precision

Each floating point value represents an estimation of the real value. When you perform operations involving such numbers the result is also an estimate. Thus expressing all digits doesn't given any more realistic answer than rounding to a small number, say 3 digits after the decimal point.

Eases the human burden of reading the data.

Reading such data back into a program means there has been a loss of precision and may not be a good thing for continuing a calculation.
Underlying Rational for **modifiers**

Like all functions, the arguments are passed as values.

But reading the values depends on knowing the data type so the proper amount of memory can be examined for the value.

The **modifiers** replace the type declaration so the function knows at read time what to look. Meaning differs with the **specifier**:  
- `d` . . . signed int  
- `hhd` . . signed char  
- `ld` . . . signed long  
- `lu` . . . unsigned long  
- `f` . . . float  
- `lf` . . . double  
- `Lf` . . . long double
Format Flags

As written, $-3.1415e+01$ has a width of 11 and precision of 4.
$3.1415e+01$ has a width of 10 and precision of 4.

Columns: 12345678901
Using %+11.4e would print: $-3.1415e+01$
                                      $+3.1415e+01$

Using % 11.4e would print: $-3.1415e+01$
                                      $3.1415e+01$

Using % -11.4e would print: $-3.1415e+01$
                                      $3.1415e+01$
**fprintf by Example**

```c
int x = 8, y = 42;
fprintf(stderr, "Step %03i produced %i\n", x, y);
```

would yield:

```
Step 008 produced 42
```

```c
double x = 81234500.0, y = 0.000125;
fprintf(stderr, "x = %7.1e; y = %9.3e\n", x, y);
```

would yield:

```
x = 8.1e+10; y = 0.125e-03
```
**sprintf by Example**

```c
int x = 8, y = 42;
char report[21];
sprintf( report, "Step %03i produced %i\n", x, y );
```

would yield the equivalent of:

```c
report[] = "Step 008 produced 42"
```
Reading with fscanf

The prototype is:

```c
int fscanf ( FILE *fh, const char *format, &var1, ..., &varN )
```

Like `fprintf()`, need to specify the file, and format string, but instead of values to output, you must provide addresses to store the read results into.

The format specifiers are **different** (simpler) than the write specifiers (again, `[` are optional elements):

```
%[*][width][modifier]specifier
```
A Plethora of Read Specification Settings

Flags:  *
Width: an integer value, else use a default length by type.
Modifiers: hh, h, (space), l, ll, j, z, t, L
Specifiers: %, c, s, [set], d, i, u, o, x, n, p
          a, A, e, E, f, F, g, G

The * flag means to skip over instead of reading and outputting a result (i.e. %*8c means skip 8 characters spaces).

See, for instance: http://en.cppreference.com/w/cpp/io/c/fscanf
Length Modifier Is Important

The a-gA-G modifiers all read and output a float. How do you read and output a double? *Modifiers* to the rescue:

\[
%f - \text{float} \\
%lf - \text{double} \\
%Lf - \text{long double}
\]

These put out 4, 8, or 16 bytes, respectively, so the addresses provided must point to the proper sized variables to hold the results. Another interesting way to cause trouble!
Width is Not Important

By default, a whitespace delimited word will be read if no width is specified. For example, consider this data line:

```
123456789 11121314
```

Reading with `%3i%3i%3i%3i%3i` would return:

```
123 456 789 11 121 314
```

But reading with `%i%i` would just the two values shown.

This can get to be tedious and error prone.
Reading with `sscanf`

The prototype is:

```c
int sscanf ( const char *s, const char *fmt, &var1, ... , &varN )
```

This basically reads a string of characters stored in memory and places the values in other locations.

```c
const char foo[] = "123456789 11121314";
long i, j;

sscanf ( foo, "%li%li", &i, &j );
```

Stores the two numbers in `i` and `j` respectively.
Return Values

`fprintf()` and `sprintf()` return the number of characters written, or a negative value if some error occurs. `sprintf()` always adds a '\0' character, which terminates strings, but does not count it.

`fscanf()` and `sscanf()` return the number of items converted, or a negative value if an error occurs.
Binary I/O

- Binary writes allow the contents of memory to be written directly to a file.
- Binary reads allow the contents of a file to be written directly into memory.
- The I/O rates are higher because there is no translation to/from text characters, and it is more compact.
- There is no loss of precision in the process. The process is, however, architecture specific!
- The data should be considered as a flow of bytes in memory storage order.
- In general, it is not readable by mere mortals.
Binary Write with fwrite()

```c
size_t fwrite( const void *buffer, size_t size, size_t count, FILE *fh );
```

- `const void *buffer` . . . Starting address in memory.
- `size_t size` . . . . . Size of the data elements.
- `size_t count` . . . . . Number of data elements.
- `FILE *fh` . . . . . . . File pointer to output file.

Why “const void”?

```c
size * count = number of bytes to write.
```
fwrite() Examples

const int dim = 1000;
int out;
double pressure[dim];
// out should be set to 1000 after this call.
out = fwrite( (const void *) pressure, sizeof(double), dim, fh );

const int dim = 1000;
int out;
double pressure[dim];
// out should likely be set to 8000 after this call.
out = fwrite( (const void *) pressure, 1, sizeof(double)*dim, fh );

const int dim = 1000;
struct xyz { float x, y, z; }
xyz coords[dim];
// out should be set to 1000 after this call.
out = fwrite( (const void *) coords, sizeof(xyz), dim, fh );
Binary Read with fread()

size_t fwrite( void *buffer, size_t size, size_t count, FILE *fh );

void *buffer . . Starting address in memory.
size_t size . . Size of the data elements.
size_t count . . Number of data elements.
FILE *fh . . . . File pointer to output file.

Why not “const void” here?

size * count = number of bytes to write.
fread() Examples

1
const int dim = 1000;
int out;
double pressure[dim];
// out should be set to 1000 after this call.
out = fread( (void *) pressure, sizeof(double), dim, fh );

2
const int dim = 1000;
int out;
double pressure[dim];
// out should likely be set to 8000 after this call.
out = fread( (void *) pressure, 1, sizeof(double)*dim, fh );

3
const int dim = 1000;
struct xyz{ float x, y, z; }
xyz coords[dim];
// out should be set to 1000 after this call.
out = fread( (void *) coords, sizeof(xyz), dim, fh );
Random Access

A handy feature of binary files is the ability to do random access on them. That is read/write to arbitrary locations in the file.

```c
long ftell ( FILE *fh );
   Reports the current location of the file pointer (bytes from beginning of the file).

long fseek ( FILE *fh, long offset, int origin );
   Positions the file pointer offset bytes from designated origin.
       SEEK_SET    . .  start of file.
       SEEK_CUR    . .  current location.
       SEEK_END    . .  end of file.
```
Random Write Example

Assume file `foo.txt` contains this string of characters (they are bytes after all): 0123456789ABCDEFGHIJKLMNOP

We could change the “ABC” to “abc” with:

```c
#include <cstdio>
#include <iostream>
int out;
FILE *fh;
char str[] = "abc";
int main ( void ) {
    fh = fopen( "foo.txt", "r+b" );
    fseek ( fh, 10, SEEK_SET );
    out = fwrite( (const void *) str, 1, 3, fh );
    std::cout << "out = " << out << std::endl;
}```
Comments on Random Write Example

`r+b` – read/write file in binary mode.

`SEEK_SET` – offset is relative to start of file.

`(const void*)` – Cast the array address to a void *ptr.

`... 1, 3, ...` – Size of 1 byte, 3 elements total.
Comments?

Questions?