Projects and Make Files

Creating an executable file requires compiling the source code into an object* file (file.o) and then linking that file with (other files and) libraries to create the executable file.

Programming projects can be big with many objects, functions, and one main program.

How to manage them? One big file? Or a bunch of smaller files.

Advantage of lots of smaller files is that you don't have to compile everything .cc—just the files that have changed. **This can save considerable time.** (The computer knows what has changed by file date.)

It also helps organize your thinking as different methods are in different files with appropriate names.

* Note object file is something different from C++ object.

Using Make

In the directory that has the project files, you create a file called ‘makefile’.

From the command line, type ‘make’, which executes the commands in the makefile. (Or for other makefile names: make -f filename)

This saves a lot of typing.

How to make a make file?

Creating object files

Part of a makefile, that compiles Movie.cu into an object file, Movie.o

```
Movie.o: Movie.cu Movie.h Vector.h
	nvcc  -c Movie.cu
```

Vector.h includes something for Movie

Format

Target: dependencies
 (Tab) command args

Example of two files

```cpp
// in myclass.cpp

class MyClass
{
public:
  void foo();
  int bar;
};

void MyClass::foo()
{
  // do stuff
}
```

```cpp
// in main.cpp

int main()
{
  MyClass a; // Compiler error: 'MyClass' is unidentified
  return 0;
}
```

Header File

```cpp
// in myclass.h

class MyClass
{
public:
  void foo();
  int bar;
};
```

```cpp
// in myclass.cpp
#include "myclass.h"

void MyClass::foo()
{
}
```

```cpp
//in main.cpp
#include "myclass.h"  // defines MyClass

int main()
{
  MyClass a; // no longer produces an error, because MyClass is defined
  return 0;
}
```

.cpp, .cu, .h, and .cuh files

Files that end in .cpp or .c or .cu contain the implementation of the methods in the problem.

Files ending in .h or .cuh that contains function prototypes, class declarations and public and private variables.

Careful of multiple header #includes

```cpp
// x.h
class X { };
```

```cpp
// a.h
#include "x.h"

class A { X x; };
```

```cpp
// b.h
#include "x.h"

class B { X x; };
```

```cpp
// main.cpp

#include "a.h"   // also includes "x.h"
#include "b.h"   // includes x.h again!  ERROR
```

Include Guards

```
//x.h

#ifndef __X_H_INCLUDED__    // if x.h hasn't been included yet...then
#define __X_H_INCLUDED__    //   #define this so the compiler knows it has been
                            // included

class X { };
…..
#endif                 //ends ifndef
```

This skips the x.h header file if already included. Use them!

Example makefile

Files: main.cpp, factorial.cpp, hello.cpp

```
all: hello

hello: main.o factorial.o hello.o
	g++ main.o factorial.o hello.o -o hello

main.o: main.cpp
	g++ -c main.cpp

factorial.o: factorial.cpp
	g++ -c factorial.cpp

hello.o: hello.cpp
	g++ -c hello.cpp

clean:
	rm -rf *o hello
```

Save as makefile

Type ‘make’ in same directory

or make clean; make

Using Macros

```
# I am a comment, CC will be the compiler to use.
CC=g++
# Comment number 2. I want to say that CFLAGS will be the
# options I'll pass to the compiler.
CFLAGS=-c -Wall

all: hello

hello: main.o factorial.o hello.o
	$(CC) main.o factorial.o hello.o -o hello

main.o: main.cpp
	$(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
	$(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
	$(CC) $(CFLAGS) hello.cpp

clean:
	rm -rf *o hello
```

MACROS

CC	compiler
DEBUG	-g if desired
CFLAGS*	compiler flags to make object files, such as -c, -Wall (warnings) -O3
LFLAGS	linker flags -lcurand, etc

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp factorial.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
   $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:
   $(CC) $(CFLAGS) $< -o $@
```

* ‘man gcc’ to see all the options for CFLAGS

Make a makefile for refdif.cu

Why? Pain to continue typing in all GL commands etc

```
nvcc -arch sm_30 -lcusparse -L/System/Library/Frameworks/OpenGL.framework/
Libraries -lGL -lGLU -Xlinker -framework -Xlinker GLUT  refdif.cu
```

makefile for refdif.cu

```
OBJ= refdif.o
LIBPATH=-L/System/Library/Frameworks/OpenGL.framework/Libraries
LDLIBS=-lcusparse -lGL -lGLU
INCPATH=
LFLAGS= -Xlinker -framework -Xlinker GLUT
CFLAGS = -D__APPLE__ -D__MACH__
CC=nvcc   $(INCPATH)

all:  refdif

refdif.o: refdif.cu
	$(CC)   -c -gencode arch=compute_30,code=sm_30  refdif.cu

refdif: $(OBJ)
	$(CC) -o refdif $(LFLAGS) $(LIBPATH) $(LDLIBS)   $(OBJ)

clean:
	rm -f *.o refdif

.PHONY: all clean
```

GPUSPH

tar file: tar -xvf filename in a directory——it will create a series of subdirectories

build	all the object files
dist	executable file (but also in top level directory)
options	saved input variables: compute_capability, problem, etc
scripts	shell scripts to look at output
src	source files and headers
tests	output from GPUSPH runs

makefile in top level directory

Source Files

Problem files: DamBreak3D.cc, DamBreakGate.cc, OpenChannel.cc, SolitaryWave.cc, TestTopo.cc, WaveTank.cc, StillWater.cc

Actual Objects: Object.cc, Cone.cc, Cube.cc, Cylinder.cc, Disk.cc, Rect.cc,
Sphere.cc, Point.cc, Vector.cc

Methods: HDF5SphReader.cc, CustomTextWriter.cc, UDPWriter.cc,TextWriter.cc,
VTKwriter.cc, VTKLegacyWriter.cc, Writer.cc

Data storage and descriptions: EulerParameters.cc, InputProblem.cc,
ParticleSystem.cc

Kernels: all .cu files (all lower case)
GPUSph.cc, forces, euler, buildneibs

Makefile

See file