USC Viterbi
School of Engineering

Compiling Options, Make, etc.

Mark Redekopp

USC

COMPILATION UNITS

Compilation Units

- Often rather than putting all our code/functions in one file it is easier to maintain and re-use code if we break them into multiple files
- We want functions defined in one file to be able to be called in another
- But the compiler only compiles one file at a time…how does it know if the functions exist elsewhere?

```
void shuffle(int items[], int len)
{
  for(int i=len-1; i > 0; --i){
    int r =  rand() % i;
    int temp = items[i];
    items[i] = items[r];
    items[r] = temp;
  }
}
```

shuffle.cpp

```
int main(int argc, char *argv[])
{
  int cards[52];

  // Initialize cards
  ...
  // Shuffle cards
  shuffle(cards, 52);

  return 0;
}
```

shuffle_test.cpp

USC Viterbi School of Engineering

Compilation Units

- We must prototype any function we want to use that is in another file
- Rather than make us type in the prototypes for each new program we write that needs that function, put prototypes in a header file that can be reused (included) for any new program

```
void shuffle(int [], int);

int main()
{
  int cards[52];

  // Initialize cards
  ...
  // Shuffle cards
  shuffle(cards, 52);

  return 0;
}
```

shuffle_test.cpp

```
void shuffle(int [], int);
```

shuffle.h

```
void shuffle(int items[], int len)
{ for(int i=len-1; i > 0; --i){
    int r =  rand() % i;
    int temp = items[i];
    items[i] = items[r];
    items[r] = temp;
} }
```

shuffle.cpp

```
#include "shuffle.h"

int main()
{
  int cards[52];

  // Initialize cards
  ...
  // Shuffle cards
  shuffle(cards, 52);
  return 0;
}
```

shuffle_test.cpp

```
#include "shuffle.h"

int main()
{ int cards[52];

  int hand[5];
  // Shuffle cards
  shuffle(cards, 52);

}
```

poker.cpp

USC Viterbi School of Engineering

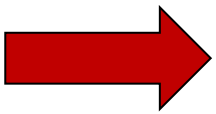
Compiling to Object Code

- Two issues:
  - We may not want to distribute our .cpp files
  - With a large program, we don't want to re-compile all the files if the code only changed in one
- Solution
  - Compiling to object code, creates the machine code/assembly code for just a single file BUT DOESN'T try to link any function calls to other files nor does it try to create an executable
  - Use: g++ –c *filename.cpp*

```
void shuffle(int items[], int len)
{
  for(int i=len-1; i > 0; --i){
    int r =  rand() % i;
    int temp = items[i];
    items[i] = items[r];
    items[r] = temp;
  }
}
```

shuffle.cpp

g++ –c shuffle.cpp

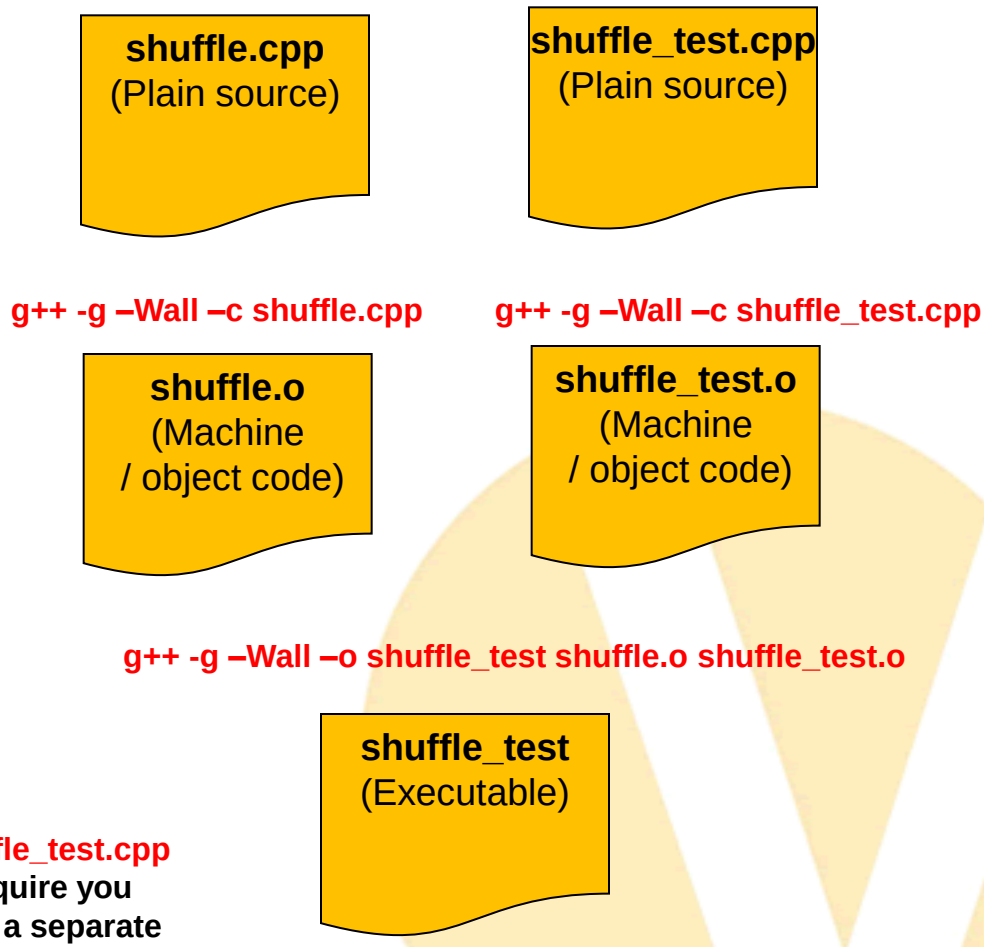


```
11001010101010101010101011
10101010101110101011101010
10111010101011...
```

shuffle.o

Linking

- After we compile to object code we eventually need to link all the files together and their function calls
- Without the –c, g++ will always try to link
- You can give g++ source files (.cpp files) or object (.o files)



Note: g++ -g –Wall –o shuffle shuffle.o shuffle_test.cpp Would also work and be fine and thus not require you to compile shuffle_test.cpp to object code in a separate step

CONDITIONAL COMPILATION

Multiple Inclusion

- Often separate files may #include's of the same header file
- This may cause compiling errors when a duplicate declaration is encountered
  - See example
- Would like a way to include only once and if another attempt to include is encountered, ignore it

```
class string{
... };
```

string.h

```
#include "string.h"
class Widget{
 public:
  string s;
};
```

widget.h

```
#include "string.h"
#include "widget.h"
int main()
{ }
```

main.cpp

```
class string { // inc. from string.h
};
class string{ // inc. from widget.h
};
class Widget{
... }
int main()
{ }
```

main.cpp after preprocessing

Conditional Compiler Directives

- Compiler directives start with '#'
  - #define XXX
    - Sets a flag named XXX in the compiler
  - #ifdef, #ifndef XXX … #endif
    - Continue compiling code below until #endif, if XXX is (is not) defined
- Encapsulate header declarations inside a
  - #ifndef XX
    #define XX
    …
    #endif

```
#ifndef STRING_H
#define STRING_H
class string{
... };
#endif
```

String.h

```
#include "string.h"
class Widget{
 public:
  string s;
};
```

Character.h

```
#include "string.h"
#include "string.h"
```

main.cpp

```
class string{ // inc. from string.h
};
class Widget{ // inc. from widget.h
...
```

main.cpp after preprocessing

Conditional Compilation

- Often used to compile additional DEBUG code
  - Place code that is only needed for debugging and that you would not want to execute in a release version
- Place code in a #ifdef XX...#endif bracket
- Compiler will only compile if a #define XX is found
- Can specify #define in:
  - source code
  - At compiler command line with (-Dxx) flag
    - g++ -o stuff –DDEGUG stuff.cpp

```
int main()
{
  int x, sum=0, data[10];
  ...
  for(int i=0; i < 10; i++){
    sum += data[i];
#ifdef DEBUG
    cout << "Current sum is ";
    cout << sum << endl;
#endif
  }

  cout << "Total sum is ";
  cout << sum << endl;
```

stuff.cpp

```
$ g++ -o stuff -DDEBUG stuff.cpp
```

COMPILER OPTIONS

g++ Options

- Most basic usage
  - `g++ cpp_filenames`
  - Creates an executable `a.out`
- Options
  - -o => Specifies output executable name (other than default a.out)
  - -g => Include info needed by debuggers like gdb, kdbg, etc.
  - -Wall => show all warnings
  - -c => compile but don't link (i.e. create an object file)
  - -I*path* => add path into #include search directory
  - -L*path* => add path into library search directory
  - -D*macro* => #define *macro*
  - -l*libname* => link in the code in library, *libname*
  - -On => n=[0..6] => Optimization level 0-6

MAKEFILES

XKCD #303

Courtesy of Randall Munroe @ http://xkcd.com

‘make’ and Makefiles

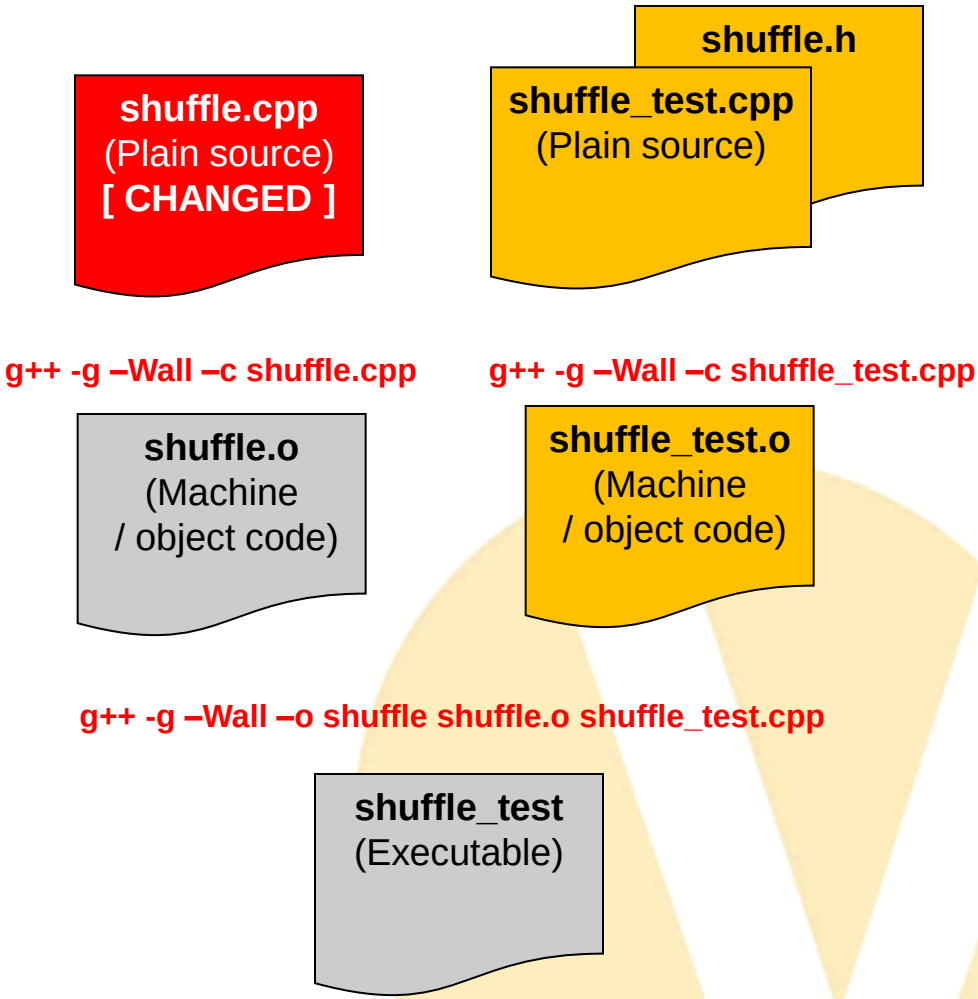
- ‘make’ is a utility program on most Linux/Unix machines which processes commands in a provided Makefile
- Helps automate compilation process
  - Essentially can use Makefiles as scripts of commands to be run
- Helps provide ‘smart’ compilation
  - Compiles only files or those that depend on files that have changed since last compilation
  - Reduces wait time for compilation process especially for large programs

Analogy: Evaluating Expressions

- Take the equation:
  - x = 5*y + (8*z + 3)
  - Evaluate for y=9, z=7
  - We evaluate term by term & add
- What if only y changed and we needed to find the new value of x? What would you re-evaluate
- What if only z changed, what operations would be needed to find the new value of x

Smart Compilation

- Only compile code that has changed and any files that DEPEND on that code



‘make’ utility

- Looks for commands in a file called ‘Makefile’ or whatever is given as the –f option on the command line
- Makefile is a text file with rules (a.k.a targets), dependencies and actions along with macros if desired

```
rule: dependencies
[TAB] action1
[TAB] action2
```

- Rules => outputs; Dependencies => Inputs; Actions => commands to build the output from inputs
- To run the Makefile, we use the make command from the command prompt:

```
make [-f filename] [specific_rule]
```

- If no target specified at command line, 'all' target is made:
- http://www.eng.hawaii.edu/Tutor/Make/

Environment variables

- From the Linux shell (terminal) we can set "variables" that contain values that can be accessed by other programs that provide system and other info
  - PATH
  - LD_LIBRARY_PATH
- Set with export command
  - export VARIABLE=VALUE
- Access value with $VARIABLE in shell
- Access value with $(VARIABLE) in Makefile
- We defined CXXFLAGS in most of your .profile or .bashrc (startup script)
  - CXXFLAGS = -g -Wall

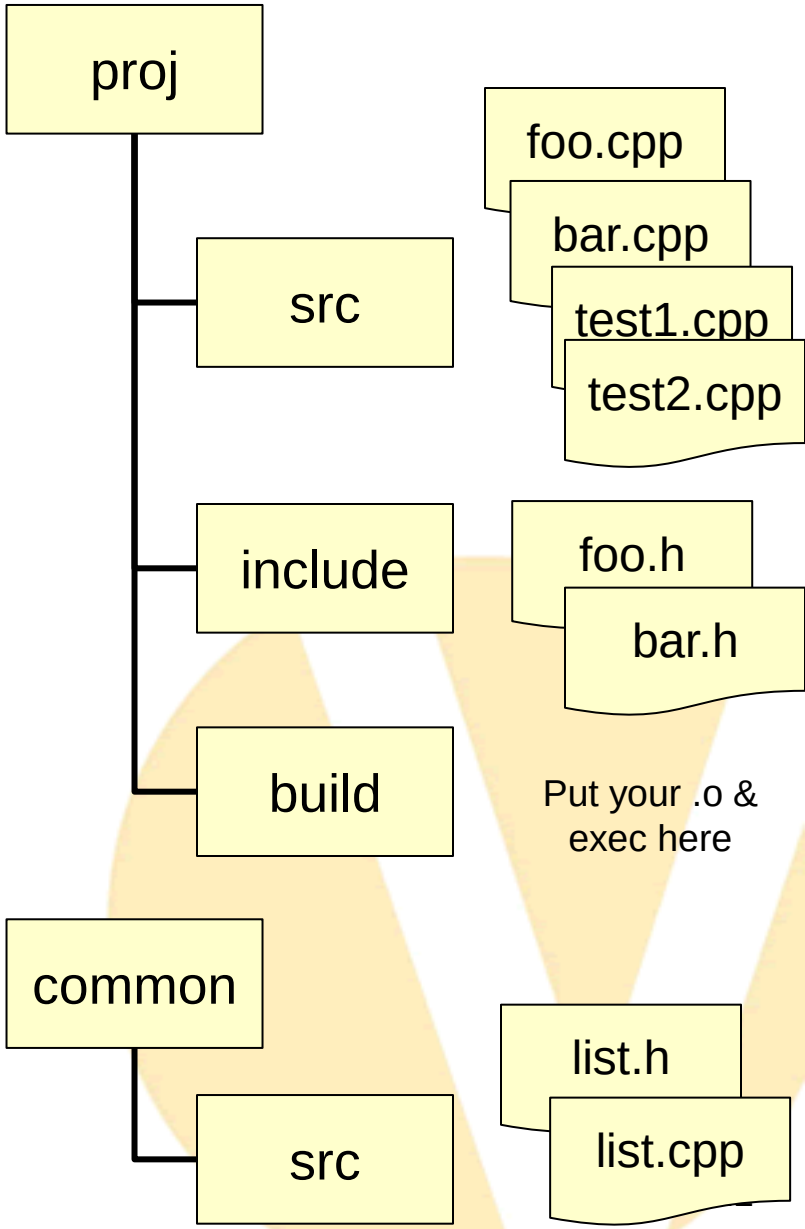
Makefile

- Defining macros/variables
  - MACRO_NAME = MACRO_DEF
  - `SRCS = test.cpp  prog1.cpp`
  - `FLAGS = $(CXXFLAGS)  # if CXXFLAGS defined by shell`
- Using macros
  - $(MACRO_NAME)
  - `$(SRCS)`
- Built-in Macros
  - `$<` = dependency name / name of the related file that caused the action
  - `$@` name of the file to be “made” / target name
- Macro Modification
  - `OBJS = ${SRCS:.cpp=.o}`
    - Substitute .o for .cpp wherever it occurs in the expansion of SRCS

USC Viterbi School of Engineering

Example

- Example
  - wget http://ee.usc.edu/~redekopp/cs104/makeex.tar
  - tar xvf makeex.tar
  - cd makeex
- test1: foo.cpp bar.cpp test1.cpp
- test2: bar.cpp list.cpp
- Two Makefile
  - Makefile.basic
    - Just hard code dependencies
    - I might suggest you go this route for your assignments…easier for now
  - Makefile.inter
    - Intermediate: uses some substitutions to be more general



USC