The Abdus Salam
International Centre
for Theoretical Physics

Compiling, Linking & Mixed Languages

Ivan Girotto – igirotto@ictp.it

Information & Communication Technology Section (ICTS)

International Centre for Theoretical Physics (ICTP)

Script Language Benefits

- Portability
  - Script code does not need to be recompiled
  - Platform abstraction is part of script library
- Flexibility
  - Script code can be adapted much easier
  - Data model makes combining multiple extensions easy
- Convenience
  - Script languages have powerful and convenient facilities for pre- and post-processing of data
  - Only time critical parts in compiled language

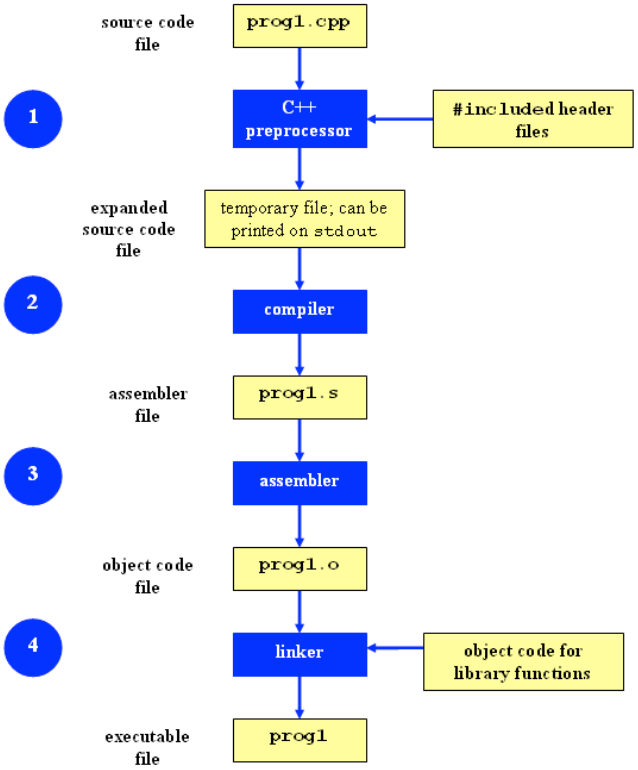
From Scripting to Compiled Codes

- maximum control of the low-level implementation
- high-performance
  - compiler are written to deliver best optimization by having full/relevant knowledge of the back-end architecture
- the O.S. loads the binary into memory and starts the execution (no other support would be required)
- direct interface to most of scientific code available

The Compiler

- Creating an executable includes multiple steps
- The “compiler” (gcc) is a wrapper for several commands that are executed in succession
- The “compiler flags” similarly fall into categories and are handed down to the respective tools
- The “wrapper” selects the compiler language from source file name, but links “its” runtime
- We will look into a C example first, since this is the language the OS is (mostly) written in

The Compiling Phases




```
#include <stdio.h>
int main(int argc, char **argv)
{
    printf("hello world\n");
    return 0;
}
```

Compilation Command examples

Pre-Processing

- Pre-processing is mandatory in C (and C++)
- Pre-processing will handle '#' directives
  - File inclusion with support for nested inclusion
  - Conditional compilation and Macro expansion
- In this case: **/usr/include/stdio.h**
  - and all files are included by it - are inserted and the contained macros expanded
- Use -E flag to stop after pre-processing:
  - **gcc -E -o hello.pp.c hello.c**
  - **cpp main.c main.i (same)**

Compiling

- Compiler converts a high-level language into the specific instruction set of the target CPU
- Individual steps:
  - Parse text (lexical + syntactical analysis)
  - Do language specific transformations
  - Translate to internal representation units (IRs)
  - Optimization (reorder, merge, eliminate)
  - Replace IRs with pieces of assembler language
- Using -S the compilation stops after the stage of compilation (does not assemble). The output is in the form of an assembler code file for each non-assembler input file specified.
  - **gcc -S hello.c (produces hello.s)**

Assembling

- Assembler (as) translates assembly to binary
  - from there, Linux tools are needed for accessing the content
- Creates so-called object files (in ELF format)
  - gcc -c hello.c
  - nm hello.o
- Be careful at *built-in* functions
  - -fno-builtin can be used to work-around the problem

Linking

- Linker (ld) puts binary together with startup code and required libraries
- Final step, result is executable
  - gcc -o hello hello.o
- The linker then “builds” the executable by matching undefined references with available entries in the symbol tables of the objects/libraries

Why is a linker interesting to us?!

- Understanding linkers will help you to build large programs
- Understanding linkers will help you to avoid dangerous programming errors
- Understanding linkers will help you how language scoping rules are implemented
- Understanding linkers will help you understand how things works
- Understanding linkers will enable you to exploit shared libraries

Object Files

- Object Files are divided in three categories:
  - Rolocatable Object Files (*.o)
  - Executable Object File
  - Shared Object Files
- Compiled object files have multiple sections and a symbol table describing their entries:
  - “Text”: this is executable code
  - “Data”: pre-allocated variables storage
  - “Constants”: read-only data
  - “Undefined”: symbols that are used but not defined
  - “Debug”: debugger information (e.g. line numbers)
- Sections can be inspected with the “readelf” command

Symbols in Object Files

```
ig@hp83-inf-21> nm visibility.o
0000000000000000 t add_abs
000000000000002a T main
                 U printf
0000000000000000 r val1
0000000000000004 R val2
0000000000000000 d val3
0000000000000004 D val4
```

```c
#include <stdio.h>

static const int val1 = -5;
const int val2 = 10;
static int val3 = -20;
int val4 = -15;
extern int errno;

static int add_abs(const int v1, const int v2) {
  return abs(v1)+abs(v2);
}

int main(int argc, char **argv) {

  int val5 = 20;

  printf("%d / %d / %d\n",
         add_abs(val1,val2),
         add_abs(val3,val4),
         add_abs(val1,val5));

  return 0;
}
```

Static Libraries

- Static libraries built with the “ar” command are collections of objects with a global symbol table
- When linking to a static library, object code is copied into the resulting executable and all direct addresses recomputed (e.g. for “jumps”)
- Symbols are resolved “from left to right”, so circular dependencies require to list libraries multiple times or use a special linker flag
- When linking only the name of the symbol is checked, not whether its argument list matches

```
#building static the library
ig@hp83-inf-21 > ar -rcs libmy.a myfile*.o


#brute force linking
ig@hp83-inf-21 > gcc main.c ./libmy.a


#Using -L (tells the compiler where look for libraries)
ig@hp83-inf-21 > gcc main.c -L./ -lmy


#Same above using gcc notation
igi@hp83-inf-21 > gcc main.c \
> -Wl,--library-path=/scratch/igirotto/linking -Wl,-lmy
```

Shared Libraries

- Shared libraries are more like executables that are missing the main() function
- When linking to a shared library, a marker is added to load the library by its “generic” name (soname) and the list of undefined symbols
- When resolving a symbol (function) from shared library all addresses have to be recomputed (relocated) on the fly.
- The shared linker program is executed first and then loads the executable and its dependencies

```
#building shared library
ig@hp83-inf-21 > gcc -shared -o mylib.so swap.o

#brute force linking
ig@hp83-inf-21 > gcc main.c ./libmy.so

#Using -L (tells the compiler where look for libraries)
ig@hp83-inf-21 > gcc main.c -L./ -lmy
ig@hp83-inf-21 > ldd a.out
   linux-vdso.so.1 =>  (0x00007fffdbb6b000)
   libmy.so => not found
   /lib64/ld-linux-x86-64.so.2 (0x00007fa003cd1000)

#Add a directory to the runtime library search
pathigi@hp83-inf-21 > gcc main.c \
> -Wl,--rpath=/scratch/igirotto/linking -Wl,-lmy
```

Using LD_PRELOAD

- Using the LD_PRELOAD environment variable, symbols from a shared object can be preloaded into the global object table and will override those in later resolved shared libraries
  - replace specific functions in a shared library
- Example: override log() with a faster version:

```
double log(double x) {
    return my_log(x);
}
```

```
$gcc -shared -o fasterlog.so faster.c -lmy_log
$LD_PRELOAD=./fasterlog.so ./myprog-with
```

How Does it Work?

- the command “make” looks for a file called [Mm]akefiles
  - you can specify a different name “make -f [makefile name]”
- This file contains a series of directives that tell the “make” utility how to compile your program and in what order
- Each file (entry) is associated with a list of other files by which it is dependent: dependency line
- If any of the associated files have been recently modified, the make utility will execute a directive command just below the dependency line

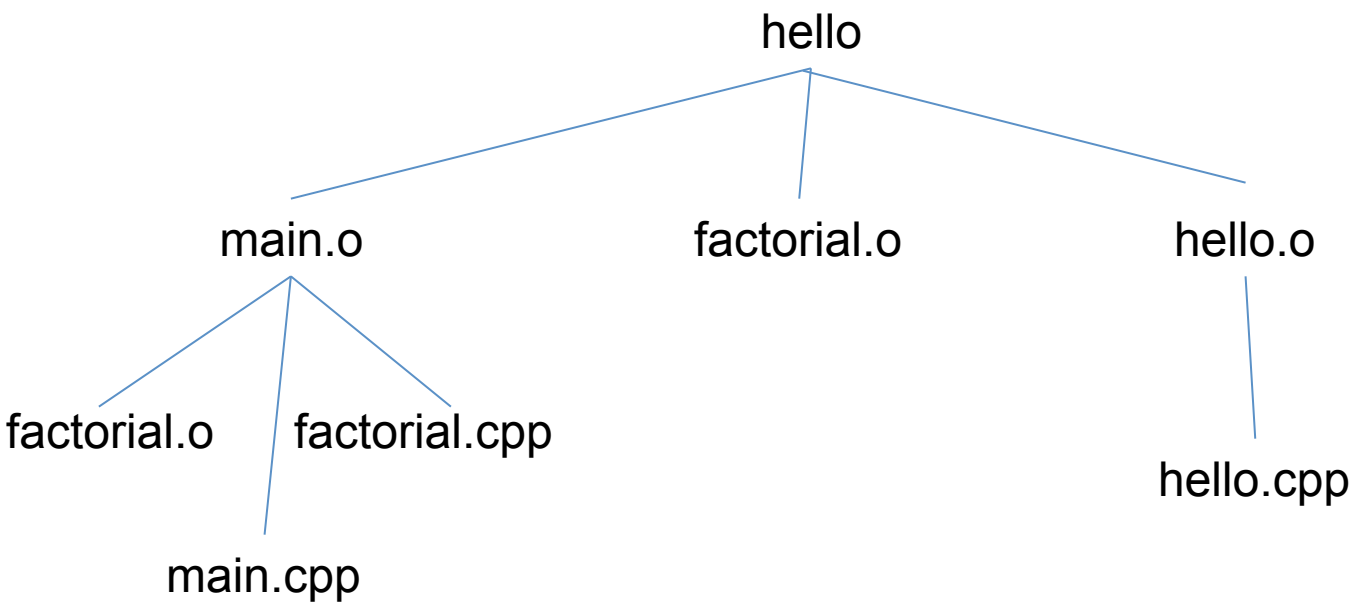
```
hello: main.o factorial.o hello.o
    g++ main.o factorial.o hello.o -o hello

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
```


hello
main.o
factorial.o
hello.o
factorial.o
factorial.cpp
hello.cpp
main.cpp

clean

Components of a Makefile

- Comments
- Rules
- Dependency Lines
- Shell Lines
- Macros
- Inference Rules

Comments

- A comment is indicated by the character "#". All text that appears after it will be ignored by the make utility until the end of line is detected.
- Comments can start anywhere. There are more complex comment styles that involve continuation characters but please start each new comment line with an # and avoid the more advanced features for now.
- Example
  - #
  - # This is a comment
  - projecte.exe : main.obj io.obj # this is also a comment.

Rules

- Rules tell make when and how to make a file. The format is as follows:
  - A rule must have a dependency line and may have an action or shell line after it. The action line is executed if the dependency line is out of date.
  - Example:

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

  - This shows hello.o as a module that requires hello.cpp as source code. If the last modified date of hello.cpp is newer than hello.o, than the next line (shell line) is executed.
  - Together, these two lines form a rule.

Dependency Lines

- The lines with a ":" are called dependency lines.
  - To the left are the dependencies
  - To the right are the sources needed to make the dependency.
- At the running of the make utility, the time and date when Project.exe was last built are compared to the dates when main.obj and io.obj were built. If either main.obj or io.obj have new dates, then the shell line after the dependency line is executed.
- The make process is recursive in that it will check all dependencies to make sure they are not out of date before completing the build process.
- **It is important that all dependencies be placed in a descending order in the file**.
- Some files may have the same dependencies. For instance, suppose that two files needed a file called bitvect.h. What would the dependency look like:
- main.obj this.obj: bitvect.h

Shell Lines

- The indented lines (**must have tab**) that follow each dependency line are called shell lines. Shell lines tell make how to build the target.
- A target can have more than one shell line. Each line must be preceded by a tab.
- After each shell is executed, make checks to see if it was completed without error.
- You can ignore this but I would not at this point.

Shell Lines

- After each shell line is executed, Make checks the shell line **exit status**.
- Shell lines that returning an exit status of zero (0) means without error and non-zero if there is an error.
- The first shell line that returns an exit status of non-zero will cause the make utility to stop and display an error.
- You can override this by placing a "-" in front of the shell command, but I would not do this.
  - Example:

    - gcc –o my my.o mylib.o

Macros

Examples of macros:

- HOME = /home/courses/cop4530/spring02
- CPP = $(HOME)/cpp
- TCPP = $(HOME)/tcpp
- PROJ = .
- INCL = -I $(PROJ) –I$(CPP) –I$(TCPP)
- You can also define macros at the command line such as
  - Make DIR = /home/faculty/whalley/public_html/cop5622proj/lib2
  - And this would take precedence over the one in the file.

Inference Rules

- Inference rules are a method of generalizing the build process. In essence, it is a sort of wild card notation.
- The "%" is used to indicate a wild card.
- Examples:
- %.obj : %.c
- $(CC) $(FLAGS) –c $(.SOURCE)
- All .obj files have dependencies of all %.c files of the same name.

Automatic variables are used to refer to specific part of rule components

```
target : dependencies
TAB    commands     #shell commands
```

```
eval.o : eval.c eval.h
   g++ -c eval.c
```

`$@` - The name of the target of the rule (`eval.o`).
`$<` - The name of the first dependency (`eval.c`).
`$^` - The names of all the dependencies (`eval.c eval.h`).
`$?` - The names of all dependencies that are newer than the target

Mixed Linking

- Fully static linking is a bad idea with GNU libc; it requires matching shared objects for NSS
- Dynamic linkage of add-on libraries requires a compatible version to be installed (e.g. MKL)
- Static linkage of individual libs via linker flags -Wl,-Bstatic,-lfftw3,-Bdynamic
- can be combined with grouping, example:
  - gcc [...] -Wl,--start-group,-Bstatic -lmkl_gf_lp64 \
    -lmkl_sequential -lmkl_core -Wl,--end-group,-Bdynamic

From C to FORTRAN

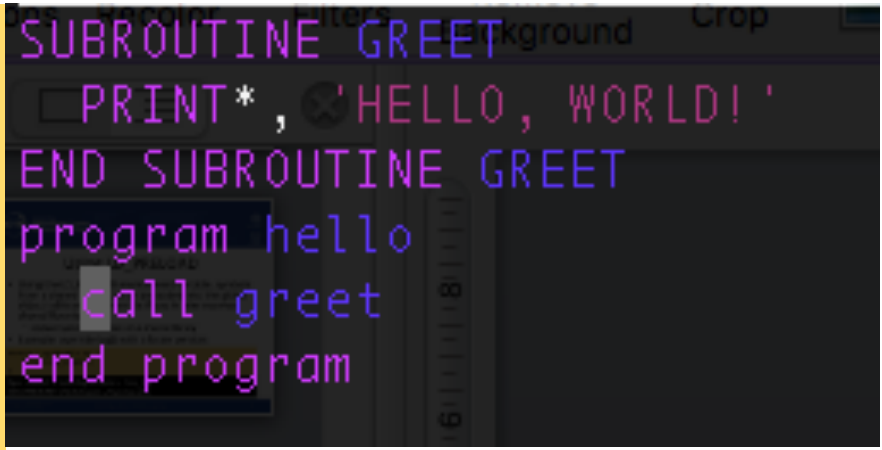
- Basic compilation principles are the same
  - preprocess, compile, assemble, link
- In Fortran, symbols are case insensitive
  - most compilers translate them to lower case
- In Fortran symbol names may be modified to make them different from C symbols (e.g. append one or more underscores)
- Fortran entry point is not “main” (no arguments) PROGRAM => MAIN__ (in gfortran)
- C-like main() provided as startup (to store args)

Pre-Processing in FORTAN

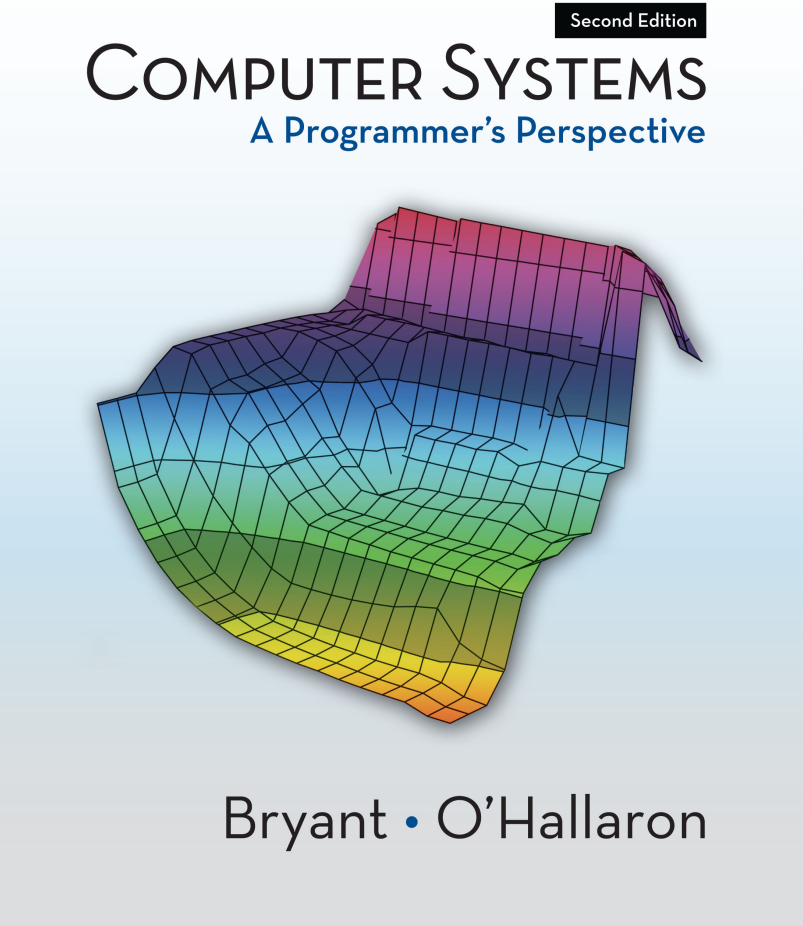
- Pre-processing is mandatory in C/C++
- Pre-processing is optional in Fortran
- Fortran pre-processing enabled implicitly via file name: name.F, name.F90, name.FOR
- Legacy Fortran packages often use /lib/cpp:
  - /lib/cpp -C -P -traditional -o name.f name.F
    - -C : keep comments (may be legal Fortran code)
    - -P : no '#line' markers (not legal Fortran syntax)
    - -traditional : don't collapse whitespace (incompatible with fixed format sources)

Symbols in Object Files (FORTRAN COMPILED)

```
ig@hp83-inf-21> nm test.o
000000000000006d t MAIN__
                 U
_gfortran_set_args
                 U
_gfortran_set_options
                 U
_gfortran_st_write
                 U
_gfortran_st_write_done
                 U
_gfortran_transfer_character_write
0000000000000000 T greet_
0000000000000078 T main
0000000000000020 r options.1.1883
```



References


Second Edition
COMPUTER SYSTEMS
A Programmer's Perspective
Bryant • O'Hallaron