

Software Development ***GCC***

Davide Balzarotti

Eurecom – Sophia Antipolis, France

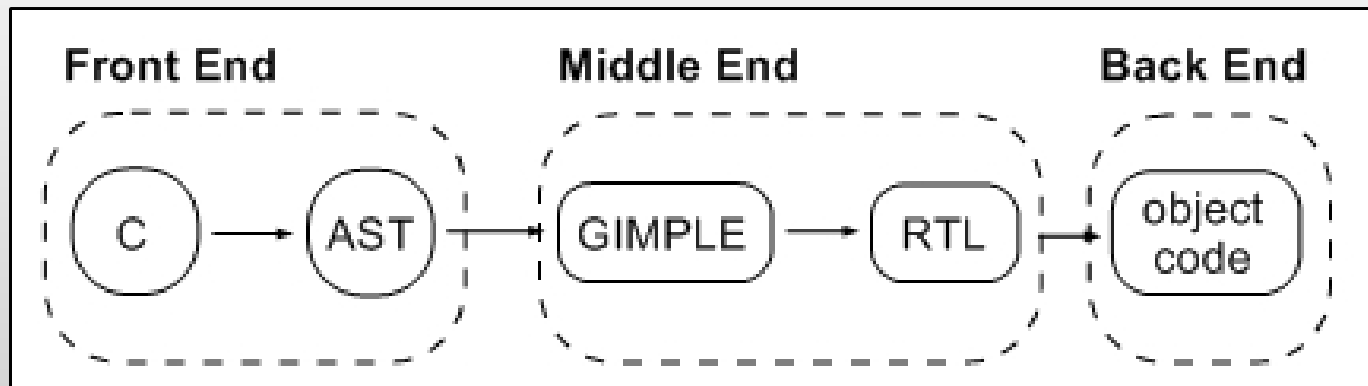
Software Development Tools

1. Writing and managing code
2. Configuring and Building the program
 - ✓ GCC
 - ✓ Makefiles
 - ✓ Autotools
3. Packaging and Distributing the application
4. Debugging and Profiling

The GNU Compiler Collection

GCC is an integrated distribution of compilers for several major programming languages

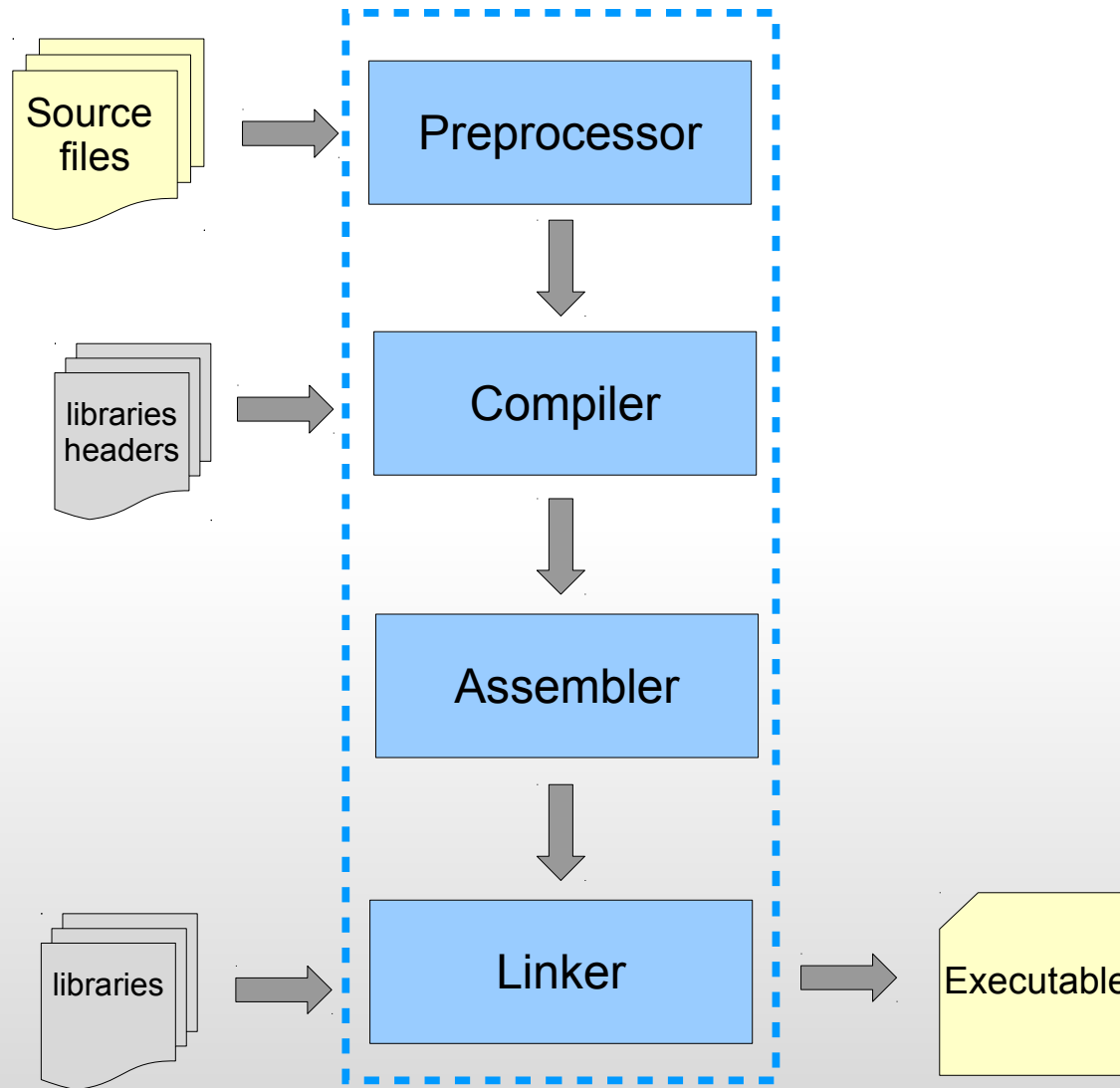
- Historically an acronym for “GNU C Compiler” (it used to support only the C language)
- **Frontends** for C, C++, Objective-C, Fortran, Java, and Ada
 - Plus several other languages that are maintained separately (e.g, for Pascal, Mercury, and COBOL)
- **Backends** can generate code for over 30 different computer architectures



GCC

- GCC is **portable**: it runs on most platforms and operating systems available today
- GCC is a **cross-compiler**: it can produce executable files for a different system from the one used by GCC itself
 - This allows software to be compiled for embedded systems which are not capable of running a compiler
- `gcc` is the program responsible to find and invoke, in the right sequence, the proper compiler, assembler and linker

A 4 Stage Process



*Use: `gcc -save-temps` to save all the intermediate files

Stage 1 - Pre-Processing

- Performed by a program called the `preprocessor` (`cpp`)
- Prepare the file for compilation (in memory)
 - Strips comments from the code
 - Places all the definitions from the included files into the `.c` source files
 - Translates all macros into inline C code
- Syntax error are not detected at this stage!
- By default, this step does not change the source file or produce any additional files.
 - `gcc -E` : execute the pre-processing phase and print the result
 - Already pre-processed file have extension “.i”

Stage 2 - Compiling

- Performed by a program called **compiler** (`cc`)
- Translates the preprocessor-modified source code into assembler code
- Checks for syntax errors and warnings
 - In case of errors, it stops and don't produce the result
 - In case of warnings, the output file is created anyway
- Requires the **header files** for all the library used by the program
- `gcc -S` : compile (and run the preprocessor if required)
 - Saves the assembler code in a “.s” file

Compiler Options (warnings)

- Options:
 - `-w` – suppress all warning messages
 - `-Werror` – transform warnings in errors
 - `-ansi` – turns off features that are incompatible with C90 or standard C++
 - `-Wall` – enables all warning for questionable code construct (implicit declarations, newlines in comments, unused functions or variables, questionable lack of parentheses, uninitialized variable usage,...)
 - `-pedantic` – check code for strict ISO conformance and generates warnings otherwise
- You can also enable/disable single warning messages
`-Wlogical-op`, `-Wconversion`, ...

Compiler Options (Optimizations)

- It is possible to enable optimizations in the assembler code by using the `-O<level>` option
- Turning on optimizations makes the compiler attempt to improve the performance and/or the code size at the expense of a longer compilation time (and possibly the ability to debug the program)
 - `-O0` straightforward compilation, no optimizations are performed
 - `-O1` use common forms of optimization that do not require too much compilation time or speed-space tradeoffs
 - `-O2` turns on almost all the optimizations, include instruction scheduling (still no optimizations requiring speed-space tradeoffs are used)
 - `-O3` turns on more expensive optimizations such as function inlining (can increase the size of the final executable)
 - `-Os` tries to generate the smaller possible executable

Compiler Options (Others)

- several fine-grained assembly options can be specified with the `-f` flag
 - `-fstack-protector` `-funroll-loops`, ...
- By default, gcc searches the following directories for header files:
 - `/usr/local/include/`
 - `/usr/include/`
- `-I path` – specify additional paths for included header files
- `-g` : tells the compiler to include debugging symbols (different format can be specified)

Stage 3 - Assembling

- Performed by a program called **assembler** (`as`)
 - `gcc -c` : execute all phases (if required) but do not invoke the linkers
 - Saves the result in a “.o” file
- The purpose of the assembler is to convert assembly language into machine code and generate an object file
 - The assembler leaves the addresses of the external functions undefined (they will be fixed later by the linker)
- Object files contains:
 - Machine code and program data
 - Relocation information - to help fix up the object code during linking
 - Symbol Table - information about symbols defined in the file and symbols to be imported from other modules
 - Debugging Information (optional)
- The format of the object file depends on the Operating System

ELF

- Executable and Linking Format (ELF) is a flexible and extensible file format for object files (executables, libraries..)
 - Currently used by Linux, *bsd, IRIX, Solaris, OpenVMS, OpenVMS, BeOS, Playstation, Wii...
 - Windows use a different format called Portable Executable (PE)
- ELF has an associated debugging format called DWARF
- ELF files contain different sections:
 - `.text` contains the executable instructions of a program
 - `.data` contains initialized data
 - `.bss` holds uninitialized data
 - `.init` `.fini` contain executable instructions for the process initialization and termination
 - ...

File Symbols

- The compiler automatically include symbols name in the object files (e.g., the name of the functions)
 - Additional debug symbols can be added by the `-g` option
- The `nm` tool can be used to print the symbols list

```
#include <math.h>
#include <stdio.h>

int global_var;
float pi = 3.1415;

int main(){
    int x = 2;
    float r;
    r = sqrt(x);
    printf("sqrt(%d)=%f\n",x,r);
    return 0;
}
```

```
> gcc -c test.c
> nm test.o
00000004 C global_var
00000000 T main
00000000 D pi
                U printf
                U sqrt
```

File Symbols

- The compiler automatically include symbols name in the object files (e.g., the name of the functions)
 - Additional debug symbols can be added by the `-g` option
- The `nm` tool can be used to print the symbols list

```
#include <math.h>
#include <stdio.h>

int global_var;
float pi = 3.1415;

int main(){
    int x = 2;
    float r;
    r = sqrt(x);
    printf("sqrt(%d)=%f\n",x,r);
    return 0;
}
```

```
> gcc -c test.c
> nm test.o
00000004 C global_var
00000000 T main
00000000 D pi
                U printf
                U sqrt
```

Symbol Type:

C	uninitialized data
D	initialized data
T	in the .text segment (code)
U	used but not defined (imported)

File Symbols

- Useful options:
 - `-S` : print symbols size
 - `--demangle` : transform back low level symbol names to user level names (very useful for C++)
- To save disk space it is possible to strip out the symbols from the binary using the `strip` command

```
> gcc -c test.c
> file test.o
test.o: ELF 32-bit LSB relocatable, Intel 80386,
version 1 (SYSV), not stripped

> strip -s test.o
test.o: ELF 32-bit LSB relocatable, Intel 80386,
version 1 (SYSV), stripped

> nm strip
nm: test.o: no symbols
```

Stage 4 - Linking

- Executed by a program called **linker** (`ld`)
- Combines the program object code with library object code to produce the final executable file
- Saves the executable code to a file
 - By default, `gcc` creates an executable program called `a.out`
 - You can specify another name with: `-o filename`
- GCC needs to find the object code for each library
- By default, it searches the following directories for libraries:
 - `/usr/local/lib/`
 - `/usr/lib/`
- `-L path` : add path to the library paths

Linking two files

func.h

```
void f(int x, int y, int z);
```

func.c

```
#include <stdio.h>
#include <func.h>

void f(int x, int y, int z){
    printf("%d\n",x+y+z);
}
```

main.c

```
#include <func.h>

int main(){
    f(1,2,3);
    return 0;
}
```

```
# compile the func file
```

```
> gcc -I. -c func.c
```

```
# compile the main file
```

```
> gcc -I. -c main.c
```

```
# link the two and generate the
# executable
```

```
> gcc -I. func.o main.o -o main
```

```
# or simply..
```

```
> gcc -I. func.c main.c -o main
```

*Note: `func.h` is included in angle brackets to show how to use linker options

Static and Dynamic Libraries

- External libraries are usually provided in two forms: static libraries (.a) and shared libraries (.so)
- **Static library** are archives of object files that are combined into the executable by the linker
 - No dependency problems: everything the program need is inside the file
- **Shared library** are not included into the executable by the linker (the library code is loaded at runtime)
 - Save disk space: the executable is much smaller
 - Save memory: one copy of the library is shared by all running programs
 - The library can be modified (e.g., to fix a bug) without having to recompile all the applications that use the library

Linking

- It is necessary to tell the linker which library we want to link to our program
- For example, if we want to link the math library (called libm):
 - `gcc calc.c /usr/lib/libm.a -o calc` → static linking
 - `gcc calc.c /usr/lib/libm.so -o calc` → dynamic linking
- Specifying the complete path of each library is too verbose
 - `-l xx` : search for a library named libxx.so or libxx.a
 - By default, gcc try to use shared libraries (if available)
If not, it uses static libraries
 - `--static` : force GCC to use the static libraries
 - The order is important, always put the `-l` options at the end
- gcc always automatically add `-lc` in order to link the standard C library (libc.so)

How do I find which Library to Link?

- Check the man pages (`man sqrt`)
- No luck? Find it yourself (command line rocks)

```
find /usr/lib /usr/local/lib -name "lib*.so"  
-exec nm -oD {} 2> /dev/null \; |  
grep " sqrt$"
```

- `-o` prints the filename associated to each symbol
- `-D` option specify that we want the dynamic symbols (i.e., we are working with a shared library)

```
find /usr/lib /usr/local/lib -name "lib*.a"  
-exec nm -os {} 2> /dev/null \; |  
grep " sqrt$"
```

- `-s` option specify that we want the symbols from a static library

- Look for:
 - T symbol: normal function definition
 - W symbol: weak definition (can be overridden by a T symbol)

Creating a Static Library

- Static libraries are just **ar** archives containing the object files
- To transform the `func.c` and `func.h` files to a static library:
 - Compile the files to object files
 - Create an archive named `libfunc.a` and add the file `func.o` to it

```
> gcc -I. -c func.c
> ar rcs libfunc.a func.o
```

- To import the library:

```
> gcc -I. -L. main.c -lfunc
```

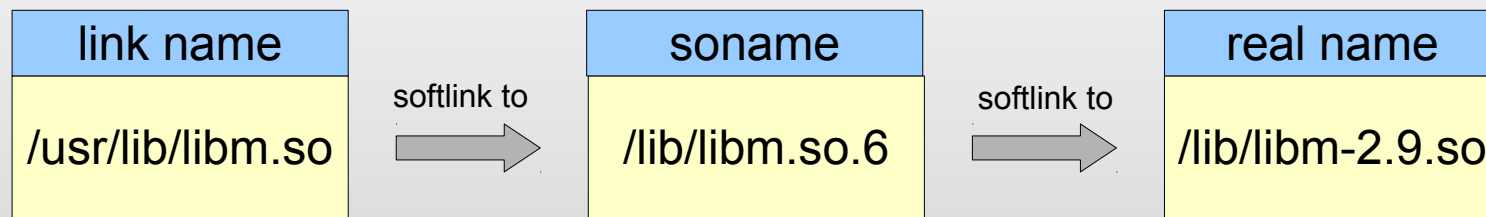
Tell the compiler where to find
the `func.h` file

Tell the linker where to find the `libfunc.a` library

Tell the linker that
we want to link a
library called `libfunc`

Shared Library - Names

- A shared library has two names: the *soname* and the *real name*
 - **soname**: `lib???.so.version` (e.g., `libattr.so.1`)
 - Version is the version number that is incremented whenever the library interface changes
 - The **real name** is the name of the file containing the actual library code (e.g., `libattr.so.1.0.1`, or `libm-2.12.1.so`)
 - The real name usually includes a minor number and a release number
- In addition, the linker usually look for a library (through the `-l` option) using yet another name



This link is created automatically by `ldconfig`

Creating a Shared Library

- First, create the object files that will go into the shared library using:
 - `-fPIC` – to generate **Position Independent Code** (required for shared libraries)
- Compile the library using:
 - `-shared` – to tell gcc that you want to create a shared library
 - `-Wl, -soname, libname` – to specify the library soname
- Put the result in `/usr/local/lib` and run `ldconfig`
- Create a symbolic link for the linker name (`libname.so`)

Creating a Shared Library

Step 1: Compile the library

```
> gcc -I. -c -fPIC func.c
> gcc -shared -Wl,-soname,libfunc.so.1
    -o libfunc.so.1.0.1 func.o -lc
```

Step 2: Install the library and Fix the names

```
> cp libfunc.so.1.0.1 /usr/local/lib
> ldconfig
> ls -l /usr/local/lib/libfunc*
... /usr/local/lib/libfunc.so.1.0.1
... /usr/local/lib/libfunc.so.1 -> libfunc.so.1.0.1
> ln -s /usr/local/lib/libfunc.so.1 /usr/local/lib/libfunc.so
```

Step 3: Compile the application

```
> gcc -I. main.c -o main -lfunc
> ldd main
    linux-gate.so.1 => (0xb804f000)
    libfunc.so.1 => /usr/local/lib/libfunc.so.1 (0xb8034000)
    libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7ed1000)
    /lib/ld-linux.so.2 (0xb8050000)
```

Shared Library at Runtime

- You can see which shared library is required by a program by using the `ldd` command
- At runtime, when the executable file is started its loader function must find the shared library in order to load it into memory
- If you place the library in a non-standard place:
 - You have to tell the linker with the `-L` option
 - You have to tell the loader by modifying the `LD_LIBRARY_PATH` environment variable

THE #1 PROGRAMMER EXCUSE
FOR LEGITIMATELY SLACKING OFF:

"MY CODE'S COMPILING."

HEY! GET BACK
TO WORK!

COMPILING!

OH. CARRY ON.

