Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

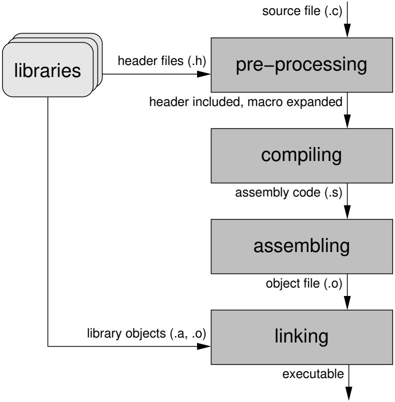
Outline

1. Understanding all 2519 options of gcc
   - Pre-processing
   - Compiling
   - Assembling
   - Linking

From C program to an executable

- A C program (which is a text file) becomes an executable after a sequence of transformations
- Each transformation takes a file as input and produces a file as output
- gcc is called the "compiler", however it makes the next 4 steps (compiling is just one step)

1. **Pre-processing**: the pre-processor syntatically replaces *pre-processor directives* (starting with "#", `#include`, `#define`, `#ifdef`, ...)
2. **Compiling**: the compiler translates the C code into assembly code
3. **Assembling**: the assembler translates assembly instructions into machine code or *object code*
4. **Linking**: object code is linked to the library code



Outline

1. Understanding all 2519 options of gcc
    - Pre-processing
    - Compiling
    - Assembling
    - Linking

Pre-processing: overview

input The original C program (text file) written by the programmer

output Another text file with all pre-processor directives being replaced/expanded (still a C program)

- The pre-processor replaces text typographically
- The "instructions" of the pre-processor are called *directives*
- Pre-processor directives starts with the symbol "#"
- Pre-processor directives are not indented: they always begin at the first character of the line
- Brief list of directives is:
  - `#define`, defines a "macro" to be replaced
  - `#include`, insert another file
  - `#if`, `#ifdef`, insert/remove portions of text depending on conditions

Pre-processing: #define directive, constants

- #define is used to define costants and macros. Classic example:

```
#define VEC_LEN 80
int v[VEC_LEN], i;

for (i=0; i<VEC_LEN; i++) {
  v[i] = /* something */;
}
```

  If VEC_LEN is changed, it is sufficient to change the value **only in one place** and not **everywhere** the length of the vector is used
- by convention macro names are always un UPPER CASE
- macros are used to configure the code
  (try `make menuconfig` to configure the Linux sources)
- a macro can be defined when invoking gcc. Example:
  `gcc -D PI=3.14` is equivalent to add at the head of file

```
#define PI 3.14
```

- Empty constants are possible: they are removed from the source file

```
#define EMPTY_CONST
```

Pre-processing: #define directive, macros

- #define can be used to define parametric macros, which may seem functions but are not!!

```
#define SQUARE(x)   x*x
a = SQUARE(2)+SQUARE(3); /* replaced by 2*2+3*3 */
```

what happens with

```
#define SUM(x,y)   x+y
a = SUM(1,2)*SUM(1,2);
```

Pre-processing: #define directive, macros

- #define can be used to define parametric macros, which may seem functions but are not!!

```
#define SQUARE(x)   x*x
a = SQUARE(2)+SQUARE(3); /* replaced by 2*2+3*3 */
```

  what happens with

```
#define SUM(x,y)   x+y
a = SUM(1,2)*SUM(1,2);
```

  it is expanded in

```
a = 1+2*1+2;     /* which is 5, not 9 !! */
```

- macro with parameters **must always** have round brackets

```
#define SUM(x,y)   ((x)+(y))
a = SUM(1,2)*SUM(1,2);
/* expanded as ((1)+(2))*((1)+(2)) */
```

Pre-processing: #define directive, long macros

- #define macros must fit in one line!
- long definitions are possible but the character \ must be used to break the line
- Example:

```
#define  EXCHANGE(type,a,b)  {\
                              type aux;\
                              aux = a; \
                              a = b; \
                              b = aux; }
```

  to be used as

  `EXCHANGE(int, a, b);`

- If `v` is a parameter of a macro, `#v` is the string of `v`. Useful for printing a variable in debugging

```
#define PRINT_INTV(v) printf("%s=%i\n",#v,v);
PRINT_INTV(var1);
/* printf("%s=%i\n", "var1", var1); */
```

Pre-processing: #include directive

- #include is used to include an external file
  - if the included file is in angular brackets
    `#include <stdio.h>`
    the file is searched in standard paths (usually \usr\include\)
  - if the included file is in double quotes
    `#include "my_header.h"`
    the file is first searched in current directory (used to include user-defined headers)
- #include is usually used to include *header files*
- A header file exports some functions of a library
- The *C standard library*, often called libc (glibc is the GNU libc) collects many useful functions
  - stdio.h, functions for input/output, files, etc.
  - string.g, string handling, copying blocks of memory
  - math.h, mathematical functions (sin, cos, pow, etc.)
  - errno.h, to test error codes set by functions
  - limits.h, architecture-dependent min/max values of different types
  - stdlib.h, random numbers, memory allocation, process control
  - ctype.h, for testing the type of characters (upper/lower case, etc.)

Pre-processing: conditional inclusion

- portions of code may be conditionally inserted by
  - “#if, #else, #endif” directives

```
#if integer-const
  /* code inserted if non-zero */
#else
  /* code inserted otherwise */
#endif
```

  - “#ifdef, #ifndef, #else, #endif” directives

```
#ifdef macro
  /* code inserted if macro is defined */
#endif
#ifndef macro
  /* code inserted if macro is not defined */
#endif
```

- conditions of #if cannot be specified by C variables!! (must be evaluated at pre-processing time, not run time)

Pre-processing: how to avoid multiple inclusions

- It may happen that a C program includes the following header files

  `#include <stdlib.h>`

  `#include <stdio.h>`

- however, they both include

  `#include <features.h>`

  which would give a “double definition” warning/error for many functions/variables

- to prevent multiple inclusions, all header file starts and ends as follows (example: `/usr/include/stdio.h`)

```
#ifndef _STDIO_H
#define _STDIO_H
/* content here */
#endif   /* _STDIO_H */
```

- try

```
gedit /usr/include/stdio.h
```

Pre-processing: temporarily removing code

for debugging purpose

- the directive #if offers a convenient way to add and remove code
- this is useful for testing purpose

```
#if 0
  /* code not inserted */
#endif
#if 1
  /* code inserted */
#endif
```

Pre-processing: pre-defined macros for debugging

- To support the debugging, the following macro are predefined

`__FILE__`	string expanded with the name of the file where the macro appears; useful with programs made by many files
`__LINE__`	integer of the line number where the macro appears
`__DATE__`	string with the date of compilation
`__TIME__`	string with the time of compilation

- A good example of debugging code is:

```
#ifdef DEBUG
#define MY_DBG printf("File %s, line %i\n",\
                      __FILE__,\
                      __LINE__)
#else
#define MY_DBG
#endif
```

Pre-processing: the NULL pointer macro

- The macro NULL represents a pointer (address in memory) which is invalid

```
#define NULL (void *)0
```

- The value of the NULL macro is zero. After

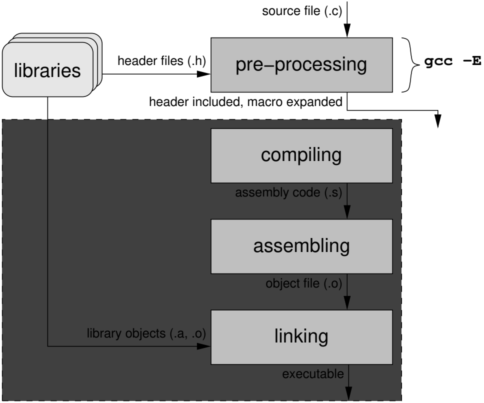
  ```
  int * p;

  p = NULL;
  ```

  all bits of the variable p are zero.

- a NULL pointers cannot be dereferenced: it does not point to any useful memory location

Pre-processing: invoking preprocessor only



- By running `gcc -E filename` the pre-processor only is executed on `filename` and the output is written to the terminal (`stdout`)

- Hence, by `gcc -E filename > after-pre-proc` the output of the pre-processor is written to `after-pre-proc` *test-preproc.c*

Using #define macro to declare standard used

- The development of C libraries and Unix is 50 years long!
- Over the years, many different libraries, standard, APIs were proposed
- *Feature Test Macros* are a way to declare the desired standard
- Examples:

```
#define _GNU_SOURCE /* recommended for SO */
#define _BSD_SOURCE
#define _POSIX_C_SOURCE
#define __STRICT_ANSI__
```

- `man feature_test_macros` or `gedit /use/include/features.h` for full description
- **Important**: the availability of some functions may depend on the these macro
  - This can be seen at the man page. Example: `man sigaction`
- These macros must appear **before** any #include directive `gedit /usr/include/stdio.h`

Pre-processing: options

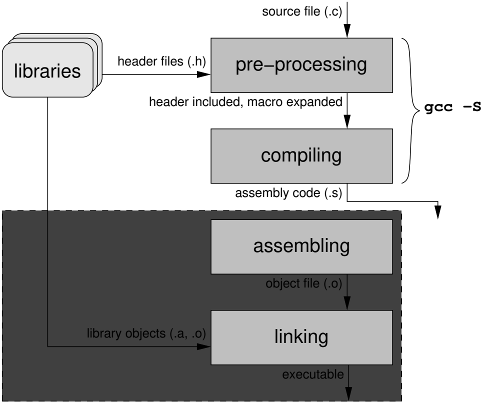
- `-E` stop after pre-processing and produce the output to the terminal (`stdout`). Must be redirected to file is it is needed to save it
- `-D` , defines a macro
- `-I <dir>`, search directory `<dir>` before standard include directories
  - useful if you want to override standard declaration of functions

Outline

1. Understanding all 2519 options of gcc
   - Pre-processing
   - Compiling
   - Assembling
   - Linking

Compiling: invoking compiler only

- After the pre-processor is run, the C program (text file) is traslated into a sequence of assembly instructions (still a text file)
- gcc can be stopped after the pre-processor and the compilation by `gcc -S ....`



- by default `gcc -S <filename>.c` saves the assembly instructions in `<filename>.s`

Compiling: options

- billions of options for compiling `man gcc`

1. Cross-compiling: produce the assembly for different architectures:
   - `-m32` 32-bit architectures
   - `-marm` ARM architectures
2. Optimization of the code
   - `-O2` some typical optimizations (such as loop unrolling): optimizations depends very much on the architecture
   - `-Os`, optimize the size of the object file
3. Debugging
   - `-g`, add debugging symbols (used by the debugger `gdb`)
   - `-O0`, no optimization (optimized code is hard to debug)
4. Try compiling by

```
gcc -S -g -O0 test-print-char.c
```

Compiling: syntax to be used for the exercises/project

1. `-std=c89`, select the ANSI C standard (the first standardized C in 1989)
   - variables are declared only at the top of the block. Not allowed to declare variables "on the fly" as in

     ```
     for (int i=0; i<10; i++) /* no C89 standard */
     ```
   - no comment

     ```
     // commento
     ```

     only

     ```
     /* commento */
     ```

     accepted
2. `-pedantic` rejects programs not conforming to the ANSI C standard

Outline

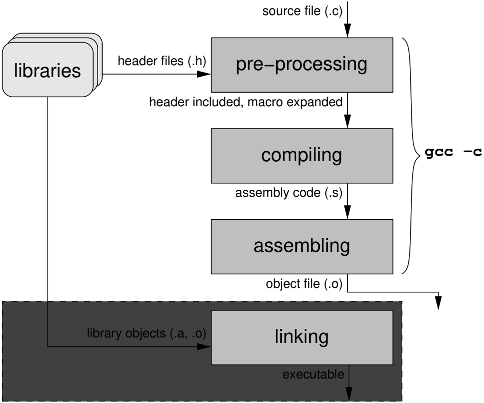
1. Understanding all 2519 options of gcc
   - Pre-processing
   - Compiling
   - Assembling
   - Linking

Assembling

- *Assembling* is the translation from the assembly instructions (still a text file readable by a text editor) into machine code (binary file, not ASCII), also called object code
- default name is `<filename>.o` (object file)
- gcc can be stopped after the assembling with `-c` option
- Try

```
gcc -c test-print-char.c
```

```
hexdump -C test-print-char.o
```

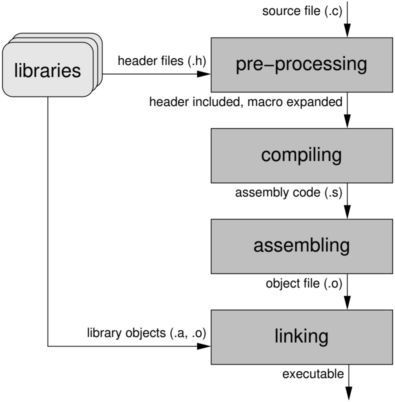


Outline

1. Understanding all 2519 options of gcc
   - Pre-processing
   - Compiling
   - Assembling
   - Linking

Linking

- Last step of `gcc` is *linking*: the pieces of code are linked together
- The linker needs one (and only one) function `main(...)` to be defined: going to be the first code to be exec
- Used to link libraries to the executable
  - GNU Standard C Library (`glibc`) linked always
  - other libraries may need explicit link, check man pages



Linking: options

- Options
  - `-L<library-path>`, search for libraries in `<library-path>` first, then in the default paths `\usr\lib`
  - `-l<lib-name>`, to link it with the library `<lib-name>`.
    Example: `-lm` to link with the math library

    ```
    man sin
    ```
  - *test-no-link.c*
    Check the difference:

    ```
    gcc -c test-no-link.c
    ```

    ```
    gcc test-no-link.c
    ```