

Environnements et Outils de Développement

Cours 3 — The C build process

Stefano Zacchioli

`zack@pps.univ-paris-diderot.fr`

Laboratoire PPS, Université Paris Diderot - Paris 7

URL <http://upsilon.cc/~zack/teaching/1112/ed6/>
Copyright © 2012 Stefano Zacchioli
License Creative Commons Attribution-ShareAlike 3.0 Unported License
<http://creativecommons.org/licenses/by-sa/3.0/>



- 1 The build process
- 2 The C preprocessor
- 3 First steps with GCC

- 1 The build process
- 2 The C preprocessor
- 3 First steps with GCC

Compiler

Definition (compiler)

A **compiler** is a computer program that transforms source code (written in some source language) into another computer language called target language.

- usually, but not always, the **source language** is a programming language that humans (programmers) are able to understand
 - ▶ e.g. C, Java, OCaml, Scala, Python, Ruby, F#, Matlab, ...
- usually, but not always, the **target language** is object code that can be executed by a hardware platform
 - ▶ e.g. x86-32, x86-64, ARM7, powerpc, etc. (native compilers)
 - ▶ e.g. JVM bytecode, Python bytecode, etc. (bytecode compilers)

General intuition

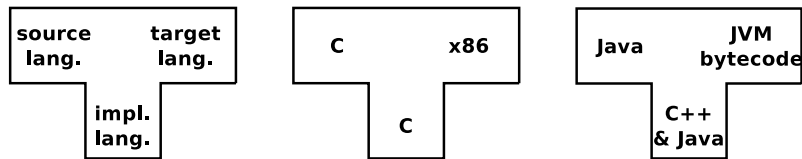
A compiler is a **translator from one language to another.**

Compiler construction

A compiler is **not a magic object**. It is a program like others :

- written in some programming language (the **implementation language**)
- by programmers like you

At least 3 languages are always involved in compiler construction : source, target, and implementation.



How do we write a compiler without a compiler?

This is the **compiler bootstrapping** problem

- ⇒ see modules *introduction à la compilation* (L3) and *compilation avancée* (M2)

luckily, we've plenty of readily available compilers these days. . .

Essential anatomy of a compiler

The architecture of a compiler consists of a few common macro-parts :

- 1 **front end** : check for **program correctness** wrt source language semantics and output an **intermediate representation** (IR) of the input program
 - ▶ e.g. lexical checking, syntax checking, type checking, etc.
- 2 **middle end** : program **optimization** by rewriting the IR
 - ▶ e.g. dead code removal, constant propagation, loop unrolling, etc.
- 3 **back end** : translate IR to the target language, doing further (platform-specific) optimization
 - ▶ e.g. **assembly language** for the target platform

Before and after compilation

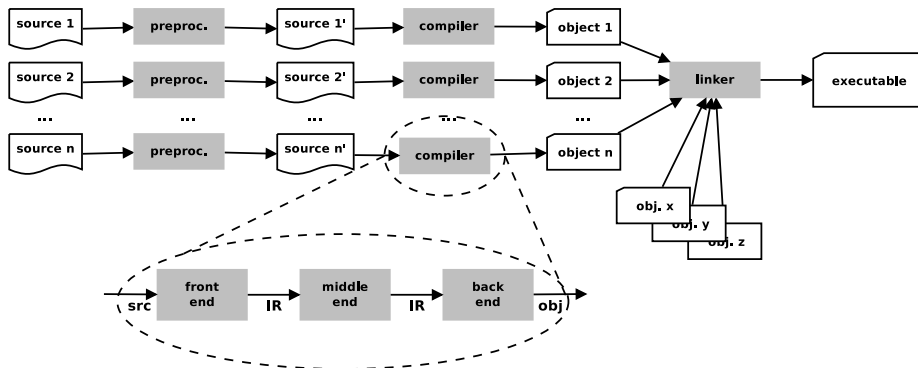
In spite of being already complex enough, we often need two more steps to get from a source to a target program.

- **Before compilation** we might have to run the source program through a **preprocessor** that prepares the program for compilation
 - ▶ e.g. the C preprocessor `cpp` takes care of all `#directives`
 - ▶ e.g. the `cam1p4` preprocessor can perform arbitrary syntactic transformation on OCaml programs
 - ▶ e.g. Lisp preprocessors, Domain Specific Languages (DSL), etc.
- **After compilation** we need to
 - 1 combine several compiled objects (i.e. the result of compiling different source files) with the needed libraries into a single compiled program
 - 2 “assemble” assembly language code to the actual sequence of bytes that the Operating system will be able to execute

A **linker** (or *link editor*) is the program that takes care of these steps

The build process

Putting it all together, the **build process** looks like this :



- general idea : each step embodies a **translation from one language to another**
- ... but the number of phases varies ; there might be more !
 - ▶ e.g. DSL → “source” code → object code → ...

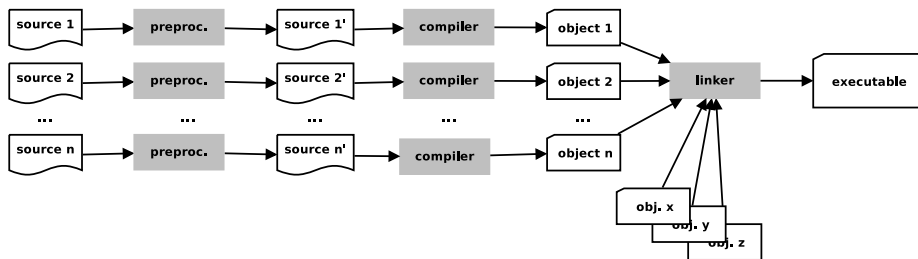
Terminology and slang — “compilation”

Strictly speaking, “compilation” is only the part of the build process in between preprocessing and linking, *extremes excluded*.

However we (as in “programmers”) often use the term “compilation” to refer to **the build process as a whole**, including preprocessing and linking.

- this slightly imprecise terminology is supported by the practice of using a **single tool to drive the entire build process**
- the tool is also usually distributed as part of compiler suites
 - ▶ `cc` — C “compiler”
 - ▶ `ocamlc` — OCaml “compiler”
 - ▶ `javac` — Java “compiler”
 - ▶ `scalac` — Scala “compiler”
 - ▶ etc.

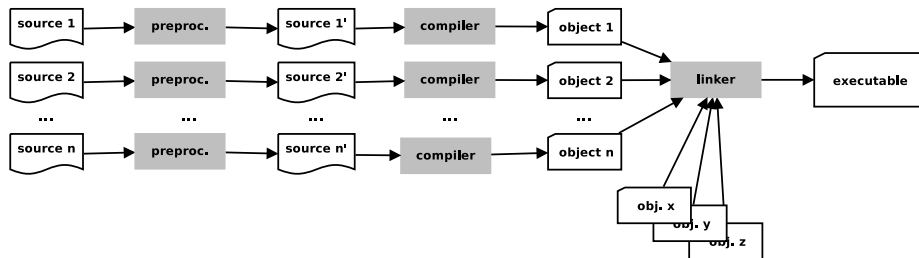
Build stages



When do the various phases happen?

- usually, **preprocessing and compilation** happens together, on a file per file basis
- **compilation and linking** might happen together (in trivial projects) or be separate phases (in medium to complex projects, i.e. almost always)

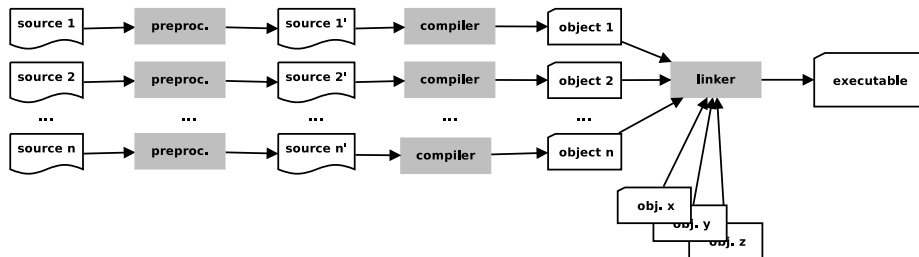
Build dependencies



- preprocessing might act on several source files at a time
 - ▶ e.g. inclusion of a header/interface file in an implementation file
 - ▶ e.g. syntactic transformations implemented as compiled programs
- to compile *source_i* we might need *object_j*, with $i \neq j$

Those are just common examples of **build dependencies**.

Build dependencies (cont.)



Typical effects of dependencies on the build process are :

- forcing a (partial) order on compilation steps
 - ▶ e.g. the linking step must be performed after the compilation steps of all involved objects
- guiding the (re-)build process to (re-)build only what is needed after only some files get changed
 - ▶ e.g. recompiling a C source file is needed only when either itself or its `#include`-s have been changed *after* the last compilation

The C build process

We are going to focus on the build process for the C language. Its architecture maps 1-1 to the one we have presented.

- the **C preprocessor** (sometimes called `cpp`) transforms C programs with `#directives` to C programs where those directives have been executed (+ line no. annotations)
- the **C linker** is the ordinary system-level linker, usually provided by the operating system
- the **C compiler** (traditionally called `cc`) transforms C programs (w/o `#directives`) to object files supported by the system-level linker
 - ▶ can also be used to drive the preprocessing and linking phase
 - ▶ we can use the C compiler as driver of the whole C build process

- 1 The build process
- 2 The C preprocessor**
- 3 First steps with GCC

The C preprocessor — generalities

The C language is defined by an [international standard](#), whose most recent incarnation is ISO/IEC 9899 :1999

- the standard supports a number of meta-language directives whose syntax is **#directive**
- the standard does not mandate the preprocessor be *a separate program*; it just defines the 4th phase of the C translation as “macro expansion and directive handling”
 - ▶ many C compilers use a [separate cpp program](#) to implement that phase
 - ▶ as the semantics of directives is [language independent](#), that allows to use the C preprocessor in other contexts

The main C language **features implemented as directives** are :

- file inclusion — `#include`
- macros — `#define`
- conditional compilation — `#if`, `#ifdef`, `#ifndef`, ...

File inclusion

File inclusion is a common feature of many text processing languages :

- an **include directive** references an **external file by name**
- the directive gets expanded to the **content of the file** as if it were included verbatim where the directive is located

the main advantage of file inclusion is **factorization** :

- 1 we can reuse the *content* of a file in different locations
- 2 while we have to maintain only one copy of it

#include

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

```
int main(void) {  
    printf("Hello, world!\n");  
}
```



```
typedef unsigned char __u_char;  
typedef unsigned short int __u_short;  
...  
extern int printf (__const char *__restrict __format, ...);  
...  
extern int scanf (__const char *__restrict __format, ...);  
...  
  
int main(void) {  
    printf("Hello, world!\n");  
}
```

#include

Where are referenced files looked for?

It depends on the used #include syntax :

- #include <file.ext> ⇒ file.ext will be looked for in the **standard compiler include path**
 - ▶ i.e. a list of pre-defined directories where to look for header files
 - ▶ can be modified using compiler switches
- #include "file.ext" ⇒ as above, but the compiler include path will be extended with the **current source directory**
 - ▶ i.e. file.ext can be in the same directory of files that want to include it

Either way, #include induces a **dependency from the including file on file.ext**

- if file.ext changes, files that include it will need to be recompiled

#include troubles

Consider the following `utils.c` :

```
#include <stdio.h>
void hello(char *msg) {
    printf("Hello %s!\n", msg);
}
```

and `hello.c` :

```
#include "utils.h"
#include "utils.h"
int main(void) {
    hello("world");
}
```

What happens when we compile `hello.c`?

?

1. multiple `#include` only *looks* stupid; it easily (and often) happens due to transitive includes

#include troubles

Consider the following `utils.c` :

```
#include <stdio.h>
void hello(char *msg) {
    printf("Hello %s!\n", msg);
}
```

and `hello.c` :

```
#include "utils.h"
#include "utils.h"
int main(void) {
    hello("world");
}
```

Error!

```
In file included from hello.c:2:0:
utils.h:3:6: error: redefinition of 'hello'
utils.h:3:6: note: previous definition of 'hello' was here
```

1. multiple `#include` only *looks* stupid; it easily (and often) happens due to transitive includes

Macros — #define

The general idea of **macros** is to define **identifiers** ↔ **content associations** (or bindings) : wherever the identifier is used, it will get replaced by the associated content by the preprocessor.

- **object-like macros** : act “like constants”
 - ▶ take no parameters ; replacement content does not depend on the invocation

Example

```
#define PI 3.14159  
circ = 2 * PI * d;
```

definition
usage

- **function-like macros** : act “like functions”
 - ▶ take parameters ; replacement content depends on them

Example

```
#define RADTODEG(x) ((x) * 57.29578)  
deg = RADTODEG(rad);
```

definition
usage

Macros — #undef

An existing macro can be **undefined** using `#undef`. The macro will not be expanded any longer in the remainder of the file.

```
#define PI 3.1415  
circ1 = 2 * PI * d1;  
#undef PI  
circ2 = 2 * PI * d2;
```



```
circ1 = 2 * 3.1415 * d1;  
circ2 = 2 * PI * d2;
```

#define expansion

```
#define PI 3.14159  
#define RADTODEG(x) ((x) * 57.29578)
```

```
circ = 2 * PI * d;  
deg = RADTODEG(rad);
```



```
circ = 2 * 3.14159 * d;  
deg = ((rad) * 57.29578);
```

Function-like macros — pitfalls

Function-like macros are powerful, but very **tricky to use!**

- for a **macro definition** to be interpreted as function like, no space should be given before the formal parameter list
 - ▶ OK **#define** RADTODEG(x) ((x) * 57.29578)
 - ▶ KO **#define** RADTODEG (x) ((x) * 57.29578)
- a function-like **macro usage** will be expanded only if it's given parameters
 - ▶ `deg = RADTODEG;` will remain unchanged (and hence likely fail)

Function-like macros — pitfalls (cont.)

- macro expansion is **language agnostic**
 - ▶ pro : can be used with other syntaxes
 - ▶ cons : you can cause syntax errors !

```
#define strange(file) fprintf (file , "%s %d",  
strange(stderr) p, 35)  
    expands to : fprintf (stderr, "%s %d", p, 35)
```

```
#define ceil_div(x, y) (x + y - 1) / y  
a = ceil_div (b & c, sizeof (int));  
    expands to : a = (b & c + sizeof (int) - 1) / sizeof (int);
```

See <http://tigcc.ticalc.org/doc/cpp.html#SEC22> for more

Best practices :

- always **balance parentheses** in macro definitions
- always put parentheses around argument usage in macro definitions

Conditional compilation

Conditional compilation is the ability to selectively compile or avoid to compile parts of the code.

- **alternative code paths** might exist in the code depending on the target platform (that is known a build-time); some of them might simply fail to compile on the wrong platform
- **optional code paths** might exist depending on the desired build
 - ▶ *development build* : with extensive debugging code and logging
 - ▶ *production build* : without any of it
- avoiding to compile unneeded optional code paths is beneficial
 - ▶ reduced **compile time**
 - ▶ reduced **object / executable size** → reduced memory usage
 - ▶ better **performances** (assert-s, if-s, ...)

Conditional compilation — #ifdef & co.

Various C preprocessor directives are used to support conditional compilation.

- #ifdef, #ifndef, #if start a conditional block
 - ▶ #ifdef and #ifndef evaluates to true, enabling the corresponding conditional block, if a **macro is defined** or not
 - ▶ #if can be used to test (a very restricted form) of **boolean arithmetic expressions** based on literal numbers and other macros
- #endif ends a conditional block (mandatory)
- #else, #elif start **alternative branches** of a conditional block
- #error can be used to **fail** at preprocessing time
 - ▶ useful when no suitable alternative compilation branch exists

Conditional compilation — examples

```
#ifdef __unix__ /* pre-defined by compilers targeting Unix */  
# include <unistd.h>  
#elif defined _WIN32 /* pre-defined by compilers targeting Win */  
# include <windows.h>  
#endif
```

```
#ifdef DEBUG  
printf("entering magic_fun\n");  
# if VERBOSE >= 2  
printf("trace message\n");  
# endif /* VERBOSE >= 2 */  
#endif /* DEBUG */
```

```
#if RUBY_VERSION == 190  
# error 1.9.0 not supported  
#endif
```

Avoiding multiple inclusion

Best practice to avoid double inclusion issues.

Sample header file `hello.h` :

```
#ifndef __HELLO_H__
#define __HELLO_H__

void hello(char *msg);

#endif /* __HELLO_H__ */
```

What would be the expansion of the following :

```
#include "hello.h"
#include "hello.h"
#include "hello.h"
```

?

The *actual* preprocessor output

```
#include <stdio.h>
int main(void) {
    printf("Hello, world!\n");
}
```



```
# 29 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
...
# 490 "/usr/include/libio.h" 3 4
extern int printf (__const char *__restrict __format, ...);
...
# 414 "/usr/include/stdio.h" 3 4
extern int scanf (__const char *__restrict __format, ...) ;
...

# 2 "hello.c" 2
int main(void) {
    printf("Hello, world!\n");
}
```

The *actual* preprocessor output (cont.)

The output of the preprocessor is intermixed with **line number annotations** — of the form `# nnn "file" . . .` — that tells the compiler where a specific line of code *really* come from.

Why is this needed?

The *actual* preprocessor output (cont.)

The output of the preprocessor is intermixed with **line number annotations** — of the form `# nnn "file" . . .` — that tells the compiler where a specific line of code *really* come from.

- the compiler check for errors wrt language semantics
- errors (& warnings) are reported to the user who should fix them
- to be meaningful to the user, **error locations** should match the files that the user is editing \neq preprocessor output
 - ▶ **line and column numbers** are affected by macro expansion

- 1 The build process
- 2 The C preprocessor
- 3 First steps with GCC**

GNU Compiler Collection — formerly “GNU C Compiler”

- one of the most widespread and reputable C compiler
- **Free Software**, released under the GNU General Public License (GPL), version 3 or above
- actually, a large **collection of compilers**
 - ▶ front-ends : C, C++, Java, Fortran, Objective-C, Ada, Go
 - ▶ back-ends : 60+, ever growing list
- support : preprocessing, compilation, linking
- releases : *1987* : 1.0 (by Richard Stallman et al.); *1992* : 2.0 ; *2001* : 3.0 ; *2005* : 4.0 ; *Oct. 2011* : 4.6.2.

man gcc

Building with gcc

All in one build :

```
$ gcc main.c
```

- preprocessing
- building
- linking
 - ▶ deliver executable a.out

(default, historical name)

Preprocessing with gcc

Preprocessing can be executed as a stand-alone phase using `cpp` :

```
$ cpp main.c
```

\$ # same, asking gcc to stop after preprocessing

```
$ gcc -E main.c
```

- will dump preprocessor output to standard output

The `-o` option can be used on all gcc (& friends) invocations to **set output destination** (overriding default names) :

```
$ cpp -o main.i main.c
```

```
$ gcc -E -o main.i main.c
```

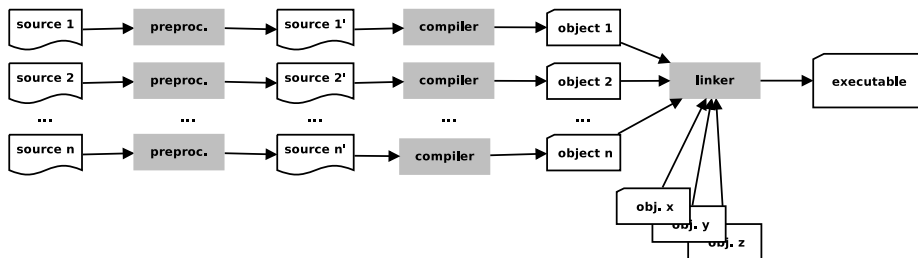
- will save preprocessor output to `main.i`

Compiling with gcc

The `-c` option asks gcc to stop after compilation

- i.e. preprocessing + compilation, but no linking

It is needed in all non trivial build processes, to compile objects separately and delay linking at the end.



Compiling with gcc (cont.)

The **default destination** for the object corresponding to a source file `source.c` is `source.o`. It can be overridden with `-o`, as usual.

It is recommended to always compile with `-Wall` that requires the compiler to **enable all warnings** about code correctness.

- uninitialized variables
- unused variables
- implicit function declaration
- missing parentheses
- etc.

Compiling with gcc — examples

```
$ gcc -Wall -c foo.c  
$ gcc -Wall -c bar.c  
$ gcc -Wall -c main.c
```

- build objects `foo.o`, `bar.o`, and `main.o`, ready to be linked

Linking with gcc

Once all objects are available, we can use gcc to link them together by simply passing them as arguments — as they were source files.

```
$ gcc -o my-program foo.o bar.o main.o
```

- will build the executable my-program linking together 3 objects
- default linking destination is a.out (if -o is omitted)

To link a program that uses **external libraries**, you will need to reference them using `-l` at link-time.

- passing `-lfoo` will tell the linker to look for the `libfoo` library in the **default library search path**

```
$ gcc -o my-program foo.o bar.o main.o -lm
```

- link with `libm` (math library)