

FROM SOURCE TO BINARY

How GNU Toolchain Works

從原始碼到二進制
ソースからバイナリへ
Von Quelle Zu Binären
De source au binaire
Desde fuente a binario
Binarium ut a fonte

Luse Cheng

Deputy Manager, Andes Technology

Jim Huang (黃敬群) <jserv@0xlab.org>

Developer & Co-founder, 0xlab

March 31, 2011 / 臺北科技大學

Rights to copy

© Copyright 2011 **0xlab**

<http://0xlab.org/>

contact@0xlab.org



Attribution – ShareAlike 3.0

You are free


- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Corrections, suggestions, contributions and translations are welcome!

Latest update: March 31, 2011

Under the following conditions

 **Attribution.** You must give the original author credit.

 **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

License text: <http://creativecommons.org/licenses/by-sa/3.0/legalcode>



《春秋》：微言大義

「隱公元年 鄭伯克段於鄆」

稱鄭伯，譏失教也

如二君，故曰克

段不弟，故不言弟

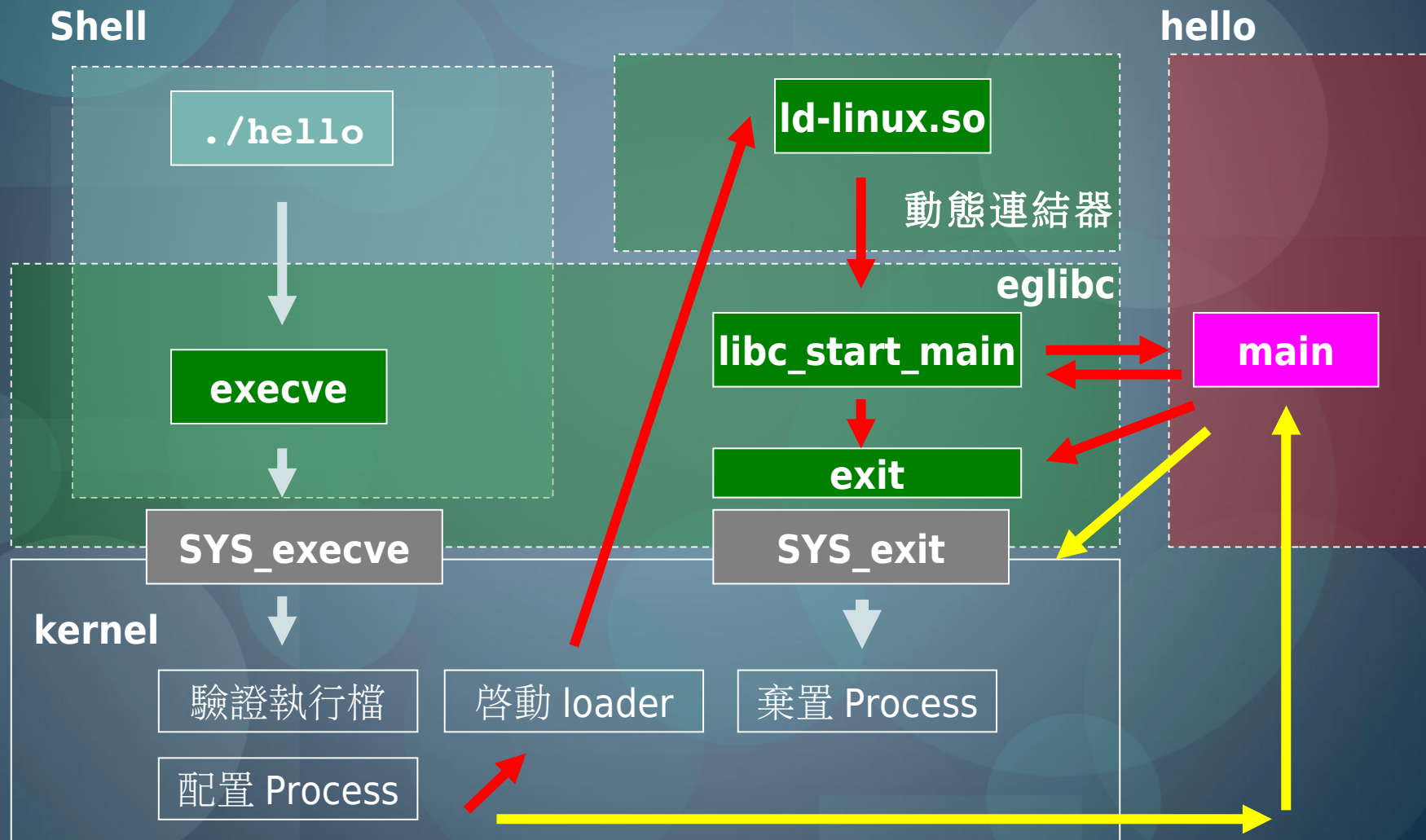
- 短短幾個字，隱含諸多涵義
- 程式員 vs. 使用者
- 了解背後的運作原理，破解微言大義

惡魔都在細節裡

- 先來觀察程式怎麼被啟動
- 文字變成程式的過程，跟程式怎麼啟動息息相關！
 - 許一個**內定**的執行環境
 - 如果很多東西都有預設環境，那考慮的事情可以減少
- 點兩下 (!?)
- # ./hello <enter>

Hello World 程式啟動流程

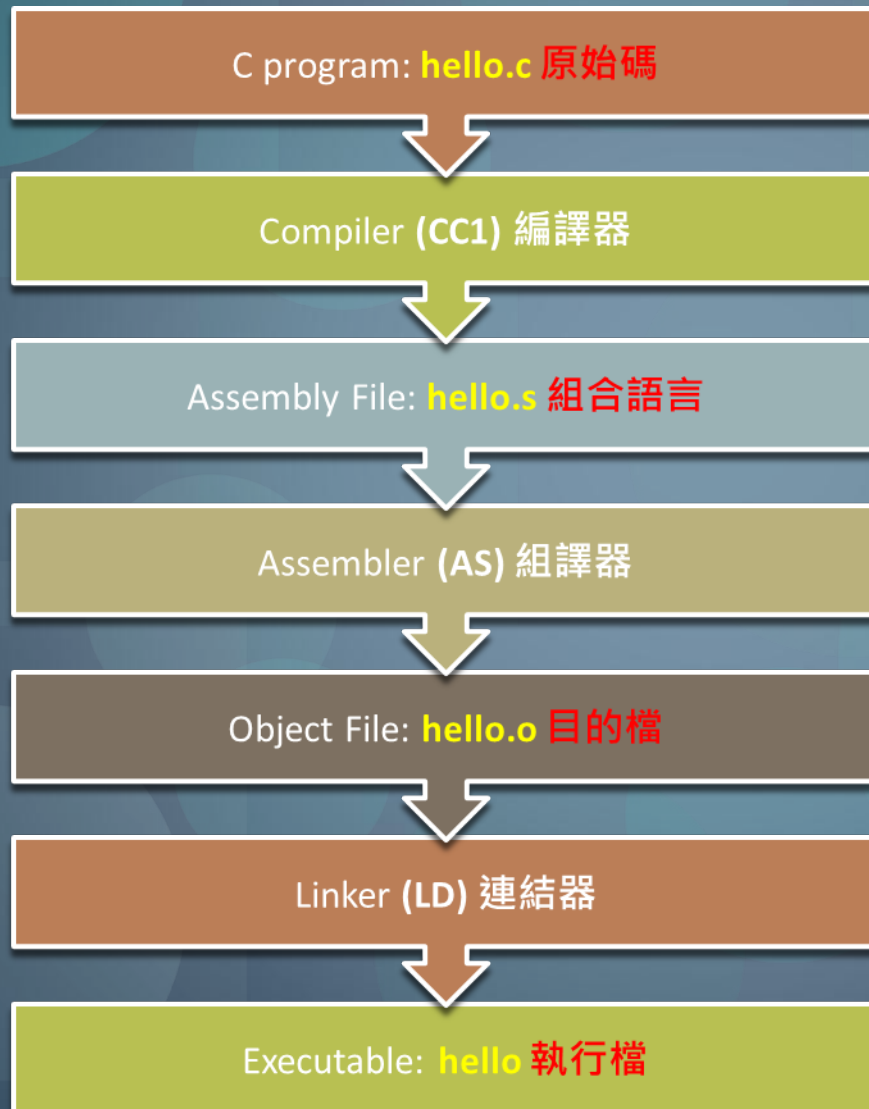
→ 動態連結
→ 靜態連結



從原始碼到二進制

- 一紙文字怎麼變成程式？
 - Ctrl + Shift + B (Visual Studio) 或 按一下「建置」
gcc hello.c -o hello
- 簡單的說就是編譯程式
- 用 Linux 來說就是 # gcc hello.c -o hello
- 微言大義：從原始碼到二進制的過程沒想像中的簡單
- 簡言之：Toolchain 就像是好人，默默的幫你把原始碼變成可執行的程式

編譯 Hello World 程式的流程



三大步驟

編譯

組譯

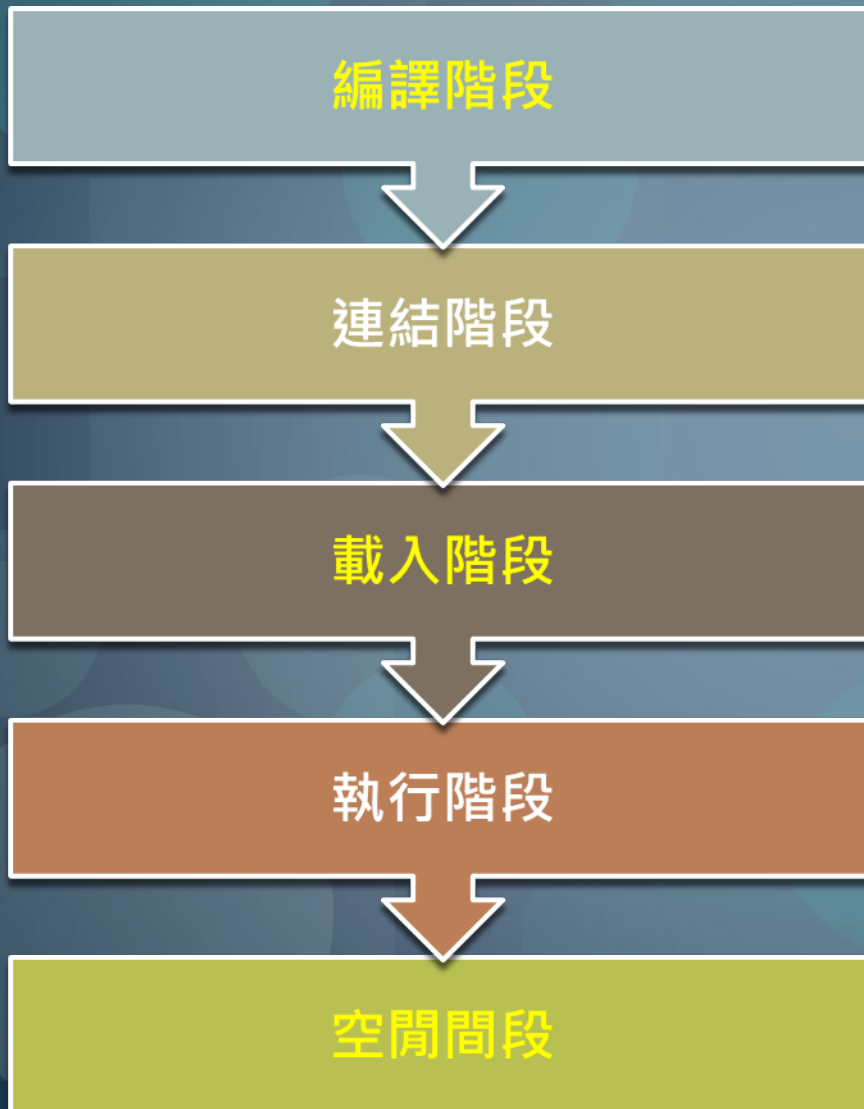
連結

對 Toolchain（好人）而言

- 除了讓服侍的對象滿意以外 ...
- 還有兩件事情很重要
 - 不能做錯事 (Quality)
 - 還要再更好 (Optimization)
- 編譯器有問題是非常讓人不能接受的
- 產生沒有效率的程式碼，也是非常讓人不能接受的

理解優化的五大階段 (Life Long Optimization)

(Chris Lattner 和 Vikram Adve, LLVM)



- 知道原始碼的結構 (O)
- 不知道變數的位址 (X)

- 不知道原始碼的結構 (X)
- 知道變數的位址 (O)

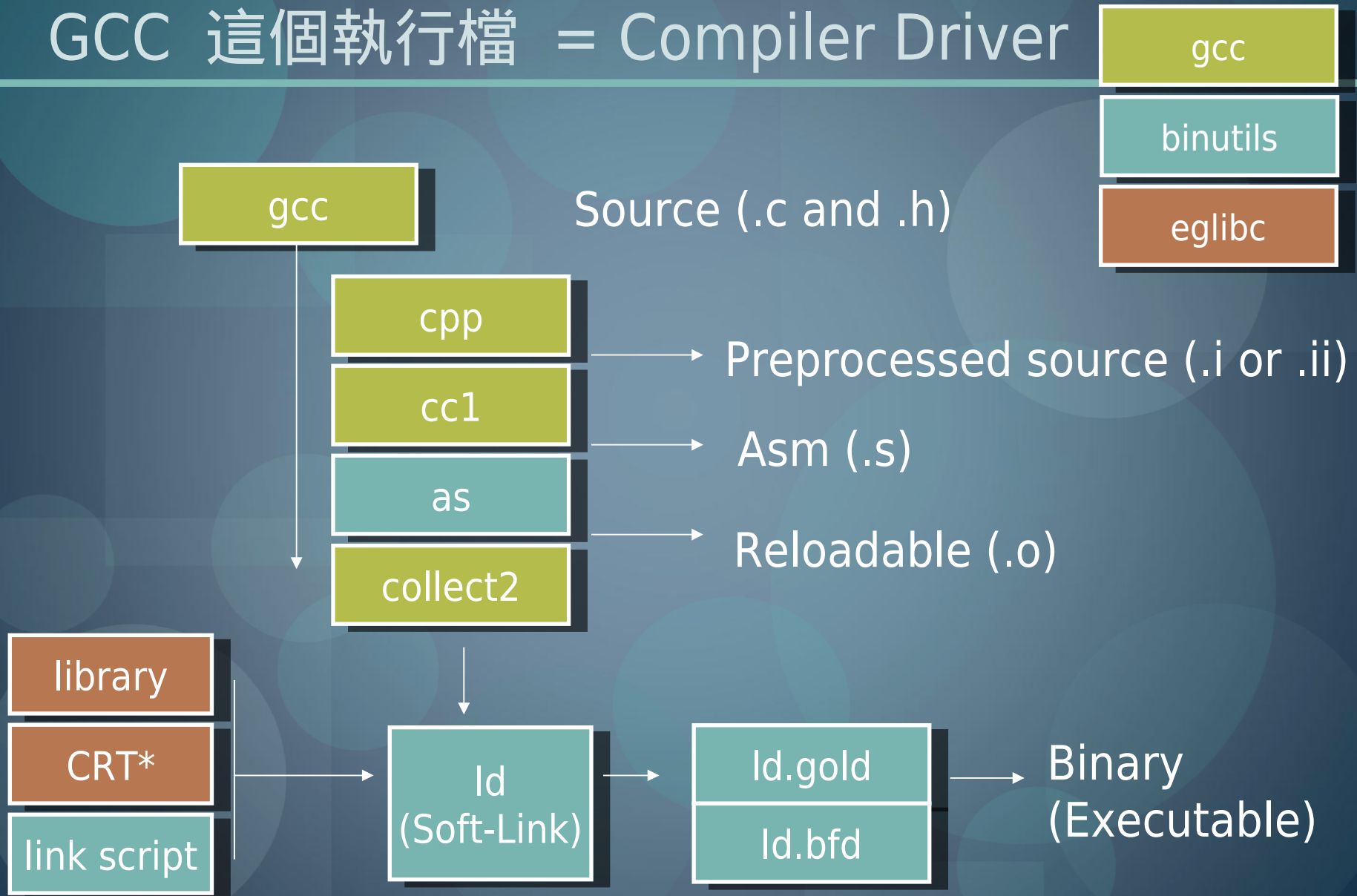
- 不知道原始碼的結構 (X)
- 不知道變數的位址 (X)
- 知道處理器的能力 (O)

正所謂是
橫看成嶺側成峰
遠近高低各不同

Compiler Driver: 好人做到底

- 不是說好 # gcc hello.c -o hello 就會有執行檔了嗎？
- 根據以上三大步驟：
- gcc 這個程式其實不只是個單純的 C 編譯器
- 再來觀察：
 - # gcc hello.c -S -o hello.s (產生組合語言檔案)
 - # gcc hello.s -o hello (從組合語言產生執行檔)
- 這樣也可以編出執行檔，難道 gcc 也是一個組譯器？

GCC 這個執行檔 = Compiler Driver



Compiler Driver 動作解析

```
# gcc hello.c -o hello -v --save-temps
```

省略路徑，只留下跟 hello.c 有關的參數，可發現：
「真正的 C Compiler 是 **cc1**」

Compiler Driver (gcc)

```
cc1 -E hello.c -o hello.i  
cc1 -fpreprocessed hello.i -o hello.s
```

編譯

```
as -o hello.o hello.s
```

組譯

```
collect2 (產生 ld 需要的參數給 ld) -o hello
```

連結

範例：ld 的參數（好人中的好人）

- collect2 version 4.4.3 (i386 Linux/ELF)

```
/usr/bin/ld --build-id --eh-frame-hdr -m elf_i386 --hash-style=both
-dynamic-linker /lib/ld-linux.so.2 -z retro /usr/lib/gcc/i486-linux-
gnu/4.4.3/../../../../lib/crt1.o /usr/lib/gcc/i486-linux-
gnu/4.4.3/../../../../lib/crti.o /usr/lib/gcc/i486-linux-
gnu/4.4.3/crtbegin.o -L/usr/lib/gcc/i486-linux-gnu/4.4.3
-L/usr/lib/gcc/i486-linux-gnu/4.4.3 -L/usr/lib/gcc/i486-linux-
gnu/4.4.3/../../../../lib -L/lib/./lib -L/usr/lib/./lib -L/usr/lib/gcc/i486-
linux-gnu/4.4.3/../../../../hello.o -v -lgcc --as-needed -lgcc_s --no-as-
needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/i486-
linux-gnu/4.4.3/crtend.o /usr/lib/gcc/i486-linux-
gnu/4.4.3/../../../../lib/crtn.o
```

GNU ld (GNU Binutils for Ubuntu) 2.20.1-system.20100303

GNU gold (GNU Binutils for Ubuntu 2.20.51-system.20100908) 1.10

想要內定卻沒那麼容易

三大法門

- **GCC** (Compiler)
- **Binutils** (Assembler, Linker)
- **libc** (C Library, eglibc/bionic)

gcc

binutils

eglibc

- 法門：修身、修心、實修
- 所以我們就要 修程式、修理論、動手下去做
- 了解三大步驟

編譯

組譯

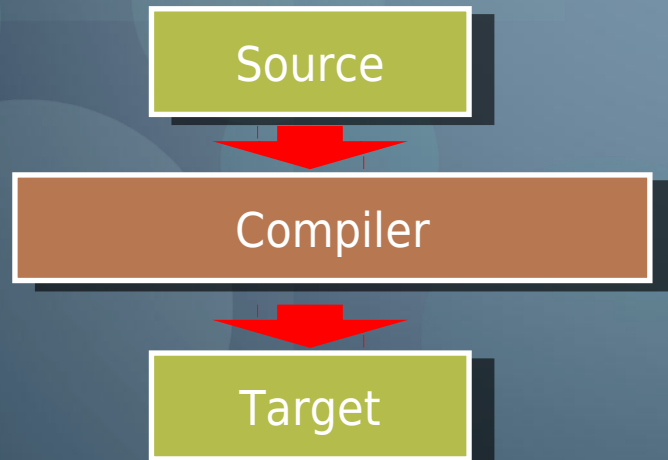
連結

第一步驟

編譯

如果有人想當你的程式碼， 那你怎樣才能做她的編譯器 (Compiler) ?

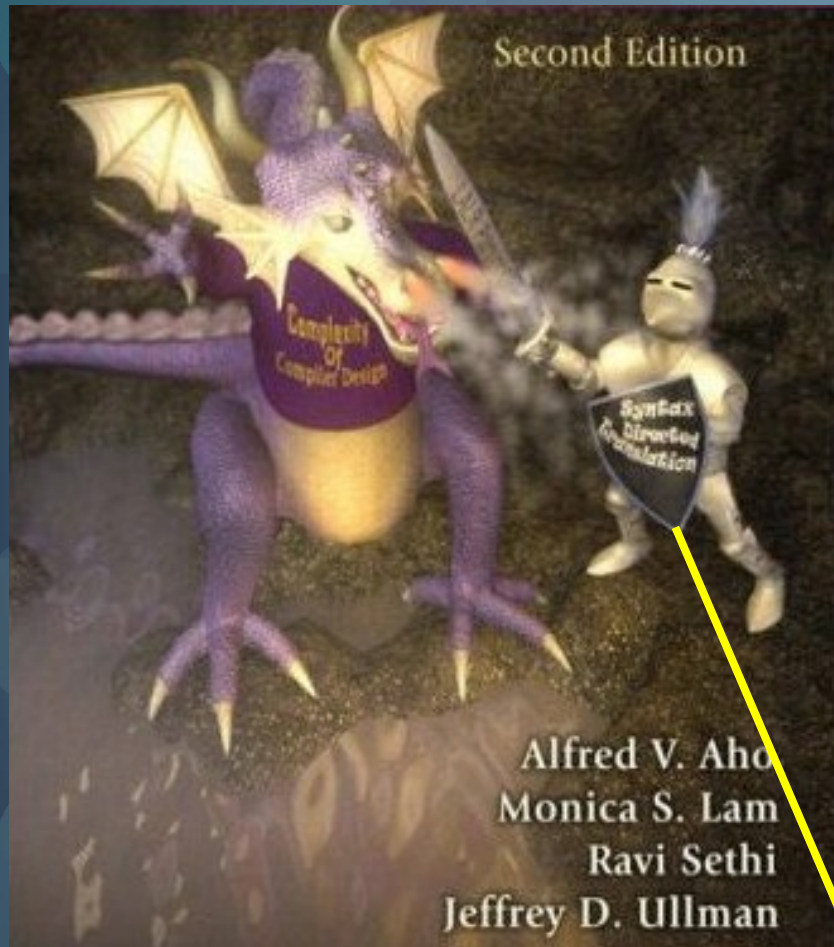
- 什麼是編譯器？
 - 翻譯機
 - Source → Target Language
- Recognizer: 讀懂她
- Translator: 轉譯她



編譯器小歷史

- First compiler (1952)
 - 第一個編譯器, UNIVAC
 - **A-0**, Grace Murray Hopper
- First complete compiler (1957)
 - 第一個完整的編譯器
 - **Fortran**, John Backus (18-man years)
- First multi-arch compiler (1960)
 - 第一個多平台編譯器
 - **COBOL**, Grace Murray Hopper et al.
- The first self-hosting compiler (1962)
 - 第一個能自己編譯自己的編譯器
 - **LISP**, Tim Hart and Mike Levin

編譯器的架構（教科書觀點）



原始碼

字彙分析（正規語言）

語法分析 (Context-Free Grammar)

語法樹

語意分析 (Type Checking ..etc)

中間語言

優化

目標語言

Syntax Directed Translator

GCC, the GNU Compiler Collection

- 海灣合作委員會 (X)



- GNU Compiler Collection (O)



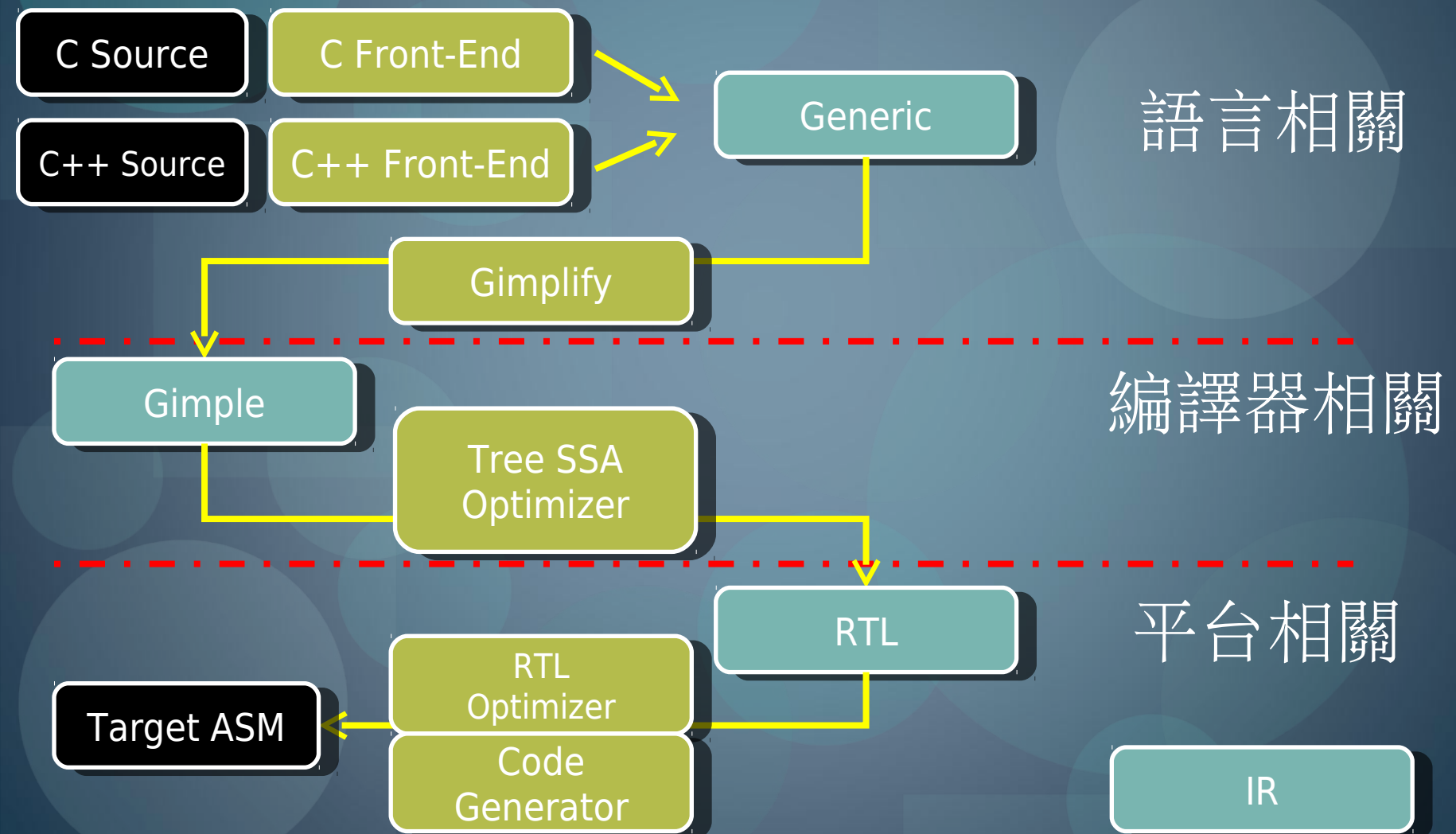
GCC – GNU Compiler Collection

- 從 GNU C Compiler 到 GNU Compiler Collection
 - 支援多種語言 (7), 多種處理器平台 (30+) (GCC 4.6.0)
 - GCC 4.6.0 支援 GO 程式語言 (GCC-GO)
 - GCC 4.6.0 超過「兩百萬行」程式碼
 - 開放原始碼編譯器平台
- **GCC** = Compiler Driver
- **cc1** = 真正的 C Compiler

GCC 的架構

- 課本終究還是課本
 - Syntax Directed Translator
 - 程式的語法樹結構將無法和處理器架構脫鉤
- GCC 的特性與挑戰：
 - GCC 支援多種語言（前端）
 - GCC 支援多種處理器（後端）
- 仔細想想：有些優化途徑與語言或硬體平台無關
 - 消除沒有用到的程式碼
 - 消除重複的運算
- 所以 GCC 引入中間層優化（編譯器相關優化）

GCC 的最終解決之道 (GCC-4 以後)



什麼是 IR (中間表示) ?

- Intermediate Representation (中間表示式)
- 編譯器的生命
 - 任何優化 / 轉換都是在 IR 上面進行的
 - IR 的重複利用性和擴展度決定一個編譯器框架會不會成功
- GCC 的 IR
 - High Level : **GENERIC** (Syntax Tree Style IR)
 - Middle Level : **Gimple** (Tree Style IR, SSA form)
 - Low Level : **RTL** (List Style IR, Register Based)
- GCC 的編譯過程：從樹狀結構 (C 語言) (Source) 到馮紐曼架構 (暫存器, 記憶體) (Binary) 的一個轉換

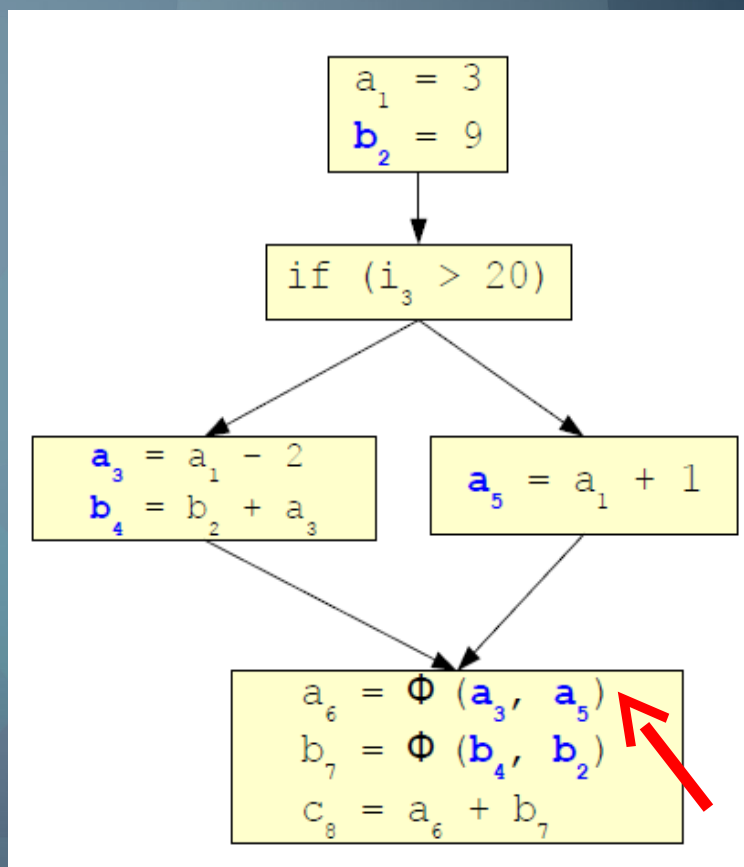
Static Single Assignment, SSA Form

- 什麼是 SSA Form ?
 - Social Security Administration (X)
 - 靜態單賦值形式 (O)
 - 用白話講，在這個表示法當中
每個被 **Assign** 的變數只會出現一次
 - 作法
 1. 每個被 **Assign** 的變數給一個 **Version number**
 2. 使用 **Φ functions**

範例：(取自 Wikipedia)

INPUT: $y = 1, y = 2, x = y$

SSA: $y1 = 1, y2 = 2, x1 = y2$



使用 SSA 可加强 / 加快以下的優化

- Constant propagation
- Sparse conditional constant propagation
- Dead code elimination
- Global value numbering
- Partial redundancy elimination
- Strength reduction
- Register allocation

SSA @ Constant propagation (常數傳遞)

- 用了 SSA 後，每次 Constant propagation 都考 100 分
 - 在 GCC-3.x GCC-4.x 間可見到顯著差異

GCC-3.x (No-SSA)

main:

...
mov r4, #0

.L5:

mov r1, #61184
add r0, r4,

#143

add r1, r1, #42
add r4, r4, #1
bl __divsi3
cmp r4, r5
ble .L5

...

```
int main()
{
    int a = 11, b = 13, c = 5566;
    int i, result;
    for (i = 0 ; i < 10000 ; i++)
        result = (a*b + i) / (a*c);
    return result;
}
```

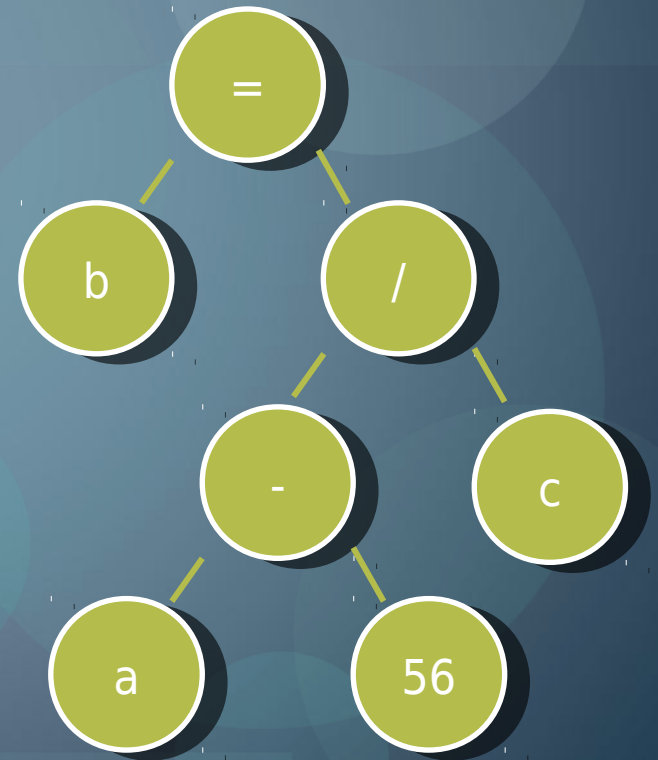
GCC-4.x (SSA)

main:

mov r0, #0
bx lr

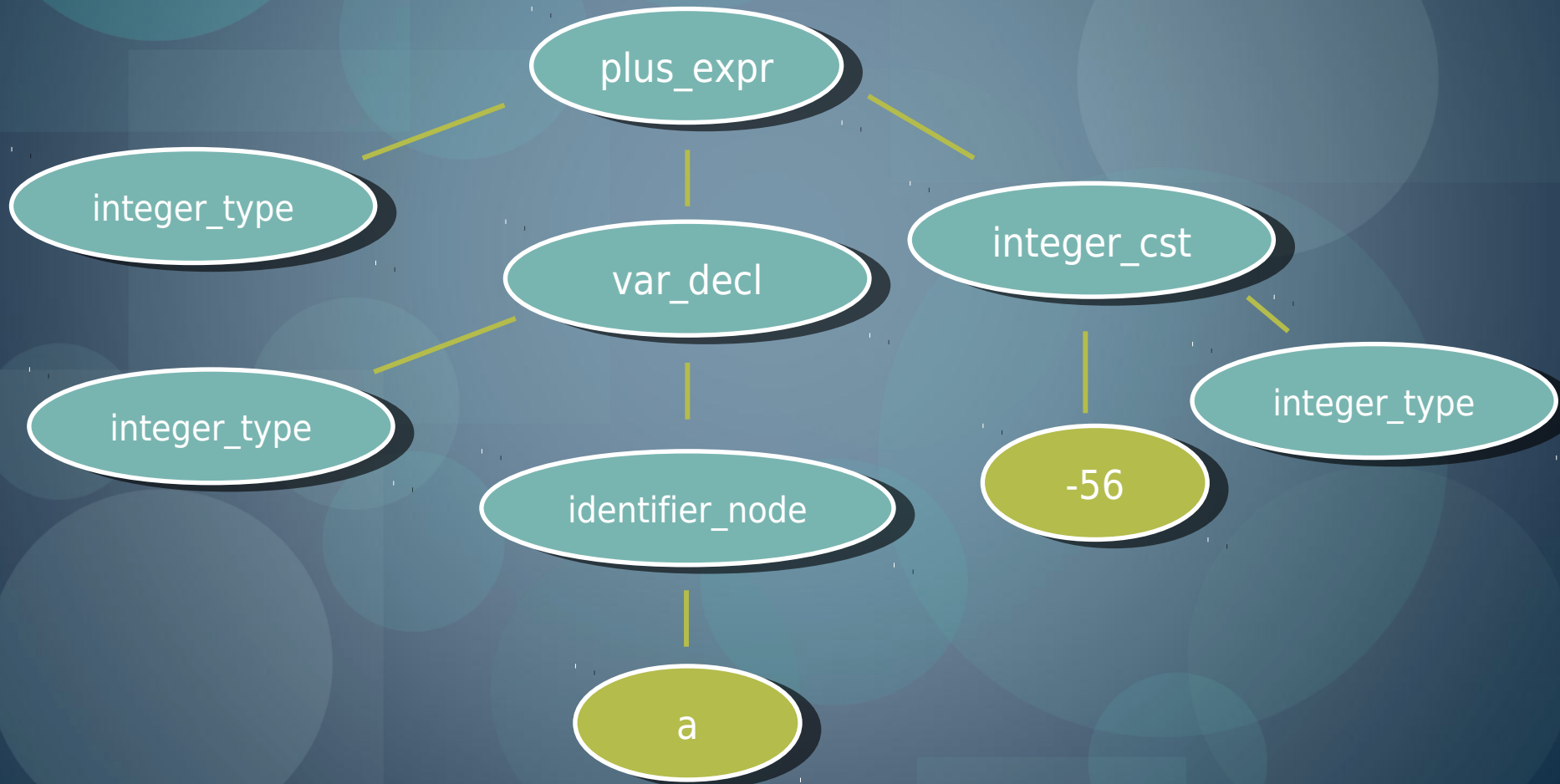
GCC 前端 : Source \rightarrow AST \rightarrow Generic

- GCC C/C++ frontend
 - BISON (3.x)
 - Recursive descent parser (4.x)
- 辨識 C 原始碼，並且轉換成解析樹 (Parsing tree)
- Abstract Syntax Tree (AST)
 - 抽象語法樹
 - 解析樹 + 語意
 - EX: $b = (a - 56) / c$



Generic Tree 範例

- (a - 56)



GCC 中端 : Gimple & Tree SSA Optimizer

- Gimple 是從 Generic Gimplify 演化而來
 - 基本上就是 Generic 然後 ...
 - 被限制每個運算只能有兩個運算元 (3-Address IR)
 - $t1 = A \text{ op } B$ (op is operator like $+ - * / \dots$ etc)
 - 被限制語法只能有某些控制流程
 - Gimple 可以被化簡成 SSA Form
 - 可使用 **-fdump-tree-`<type>`-`<option>`** 來觀看其結構
- Tree SSA Optimizer
 - 100+ Passes
 - Loop, Scalar optimization, alias analysis ...etc
 - Inter-procedural analysis, Inter-procedural optimization

GCC 後端 : Register Transfer Language (RTL)

$b = a - 56$

```
(set (reg:SI 60 [b])  
      (plus:SI (reg:SI 61 [a])  
                (const_int -56 [0xffffffffc8]))))
```

- LISP-Style Representation
- RTL Uses Virtual Register (無限多個 Register)
- GCC Built-in Operation and Arch-defined Operation
- Instruction scheduling (Pipeline scheduling)
- Peephole optimizations

GCC 後端 : RTL Patten Match Engine

MIPS.md

```
(set (reg:SI 60 [b])  
      (plus:SI (reg:SI 61 [a])  
                (const_int -56 [0xffffffffc8]))))
```

```
(define_insn "*addsi3"  
  [(set (match_operand:GPR 0 "register_operand" "=d,d")  
        (plus:GPR (match_operand:GPR 1 "register_operand" "d,d")  
                  (match_operand:GPR 2 "arith_operand" "d,Q")))]
```

"!TARGET_MIPS16"

"@"

addu\t%0,%1,%2

addiu\t%0,%1,%2

```
[(set_attr "type" "arith")  
 (set_attr "mode" "si")]
```

指令的屬性 (用於
pipeline scheduling)

d 代表是 Register
Q 代表是整數

$b = a - 56$

指令限制 (範例表示非 MIPS16 才適用此定義)

addiu \$2, \$3, -56

GCC 內建函式 (Built-in Function)

- GCC 為進行優化，會辨認標準函式，若符合條件，即可用處理器最適合的方式來處理
 - strcpy() → x86 字串處理指令
 - 無需處理格式的 printf() → puts()
 - 三角函數 → sincos (x87 指令)
- 其他難以用 C 語言表達的指令，也可製造內建函式讓使用者可以像 C 語言一般使用 (intrinsic)
 - 如：Intel MMX, ARM NEON, data prefetch, ... etc.

C 語言函式庫

- 實做：newlib, eglibc, bionic libc
- 標準函式庫
 - libc (C 語言函式庫)
 - libm (數學函式庫)
- 非標準函式
 - POSIX 系統函式庫
 - 作業系統相關函式庫
 - 網路，加解密，字元碼 .. 等雜項
- Threading Library (執行緒函式庫)
- Dynamic Linker (動態連結器)

可是處理器不支援除法指令

- 那怎麼辦？！
 - 那只好用減法和其他運算模擬了
- GCC 某些內建的運算（如除法），可以對映到 Runtime Library 的一個輔助函式
- **libgcc**: GCC low-level runtime library
 - 用來彌補處理器不足的基本運算
 - 32 位元整數除法
 - 64 位元整數加減乘除
 - 浮點運算
 - 用來彌補語言的特性
 - Exception handling

相信你的編譯器

- **Linux-Kongress 2009: Source Code Optimization**

http://www.linux-kongress.org/2009/slides/compiler_survey_felix_von_leitner.pdf

- 在 x86 使一個暫存器內含值清為 0

● <code>mov \$0,%eax</code>	b8 00 00 00 00
● <code>and \$0,%eax</code>	83 e0 00
● <code>sub %eax,%eax</code>	29 c0
● <code>xor %eax,%eax</code>	31 c0

- 哪道指令最好？

第二步驟

組譯

Binutils – GAS (GNU Assembler)

- 組譯器 (Assembler)
 - 組合語言翻成目的碼 (object file)

- 組譯器可以很簡單、也可很複雜

- 一對一翻譯 vs. 支援超多 Macro
- directives 和 pseudo-instructions

- 範例：MASM 的 invoke

```
Invoke MessageBox, 0, ptr "World", ptr "Hello", 1
```

- ```
push 0x1
push ptr "Hello",
push ptr "World"
push 0x0
call MessageBox
```

- 可根據 MessageBox 原型 (prototype) 自動修改

- Assembly Language 也可以很像 C

- Blackfin DSP

# 其實你不懂組譯器的心

- 其實組譯器沒有想像中簡單

```
binutils-2.21$ wc -l gas/config/tc-arm.c
23677 gas/config/tc-arm.c
```

- 一個 ARM Assembler 不含共用的程式碼 (BFD)、扣除 header 檔，竟要 2 萬 3 千多行 !!

# Assembler: 位址處理和 Relocation

- 位址處理

- 案例：如何將 ARM 組合語言 `bl printf` 轉成目的碼？
  - 就目前而言，組譯器不知道 `printf` 在執行檔 / 記憶體有位址
  - 連結階段才會知道 `printf` 真正的位置（靜態連結）

- Relocation Type

- 指導棋（給 Linker 的一種位址運算）
- Linker 會依造命令做絕對值，做加法（相對），特殊運算
- Dynamic Linker 也需要指導棋

- 其實組譯器除了翻譯指令以外，另外一個重要的工作就是產生 Symbol Table 跟 Relocation Table

# Assembler: Symbol Table 和 Relocation Table

bl printf

ARM Assembler

|    |     |                         |      |
|----|-----|-------------------------|------|
| 位址 | 8:  | ebffffffe bl 0 <printf> | 目的碼  |
| 位址 | 8:  | R_ARM_CALL printf       | 重定表格 |
| 序號 | <1> | printf                  | 符號表格 |

還沒有被解決 (resolve) 的位址，通常會先填 0



# Assembler: 更複雜的位址處理

- 思考一個問題：

- 32 位元 RISC 的函式呼叫指令

- 不可能涵蓋 32 位元的空間 (opcode + operand)
    - ARM BL,BLX →  $\pm 32\text{MB}$  的涵蓋範圍
    - 但 printf 可能在 32 位元中的任何位置

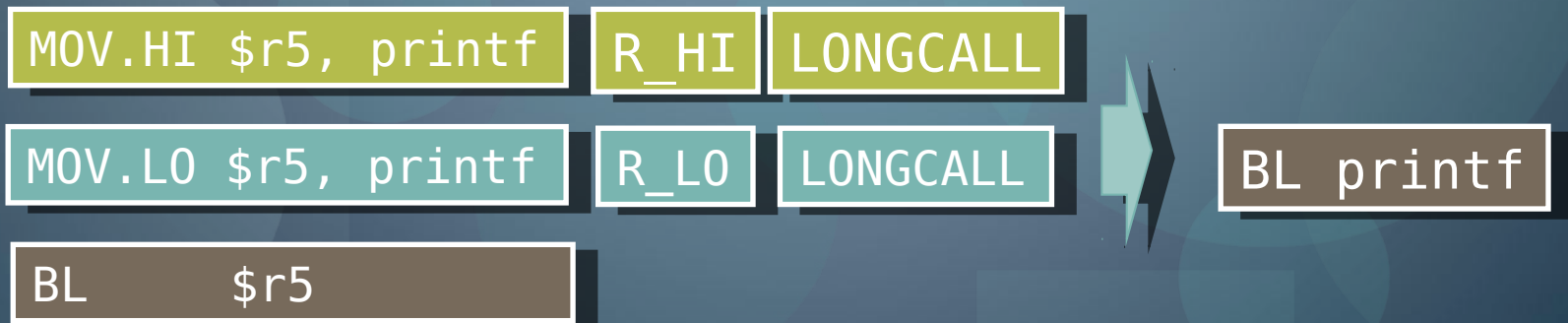
- 解法

- 把 printf 的位址當成 data 來保存
  - 保守作法：先產生 printf 的位址到暫存器

```
MOV.hi $r5, printf
MOV.lo $r5, printf
BL $r5
```

# Assembler: 連結階段最佳化

- 思考保守作法：先產生 printf 的位址到暫存器
  - 造成時間和空間的浪費
- 小心 printf 就在你身邊
- 用組譯器來達成連結階段最佳化
  - 先假設所有的位址都是 32 位元（保守作法）
  - 下「若 printf 就在附近，就消除位址運算」這個指導棋 (Relocation Type)



## 第三步驟

連結

# Binutils – collection of binary tools

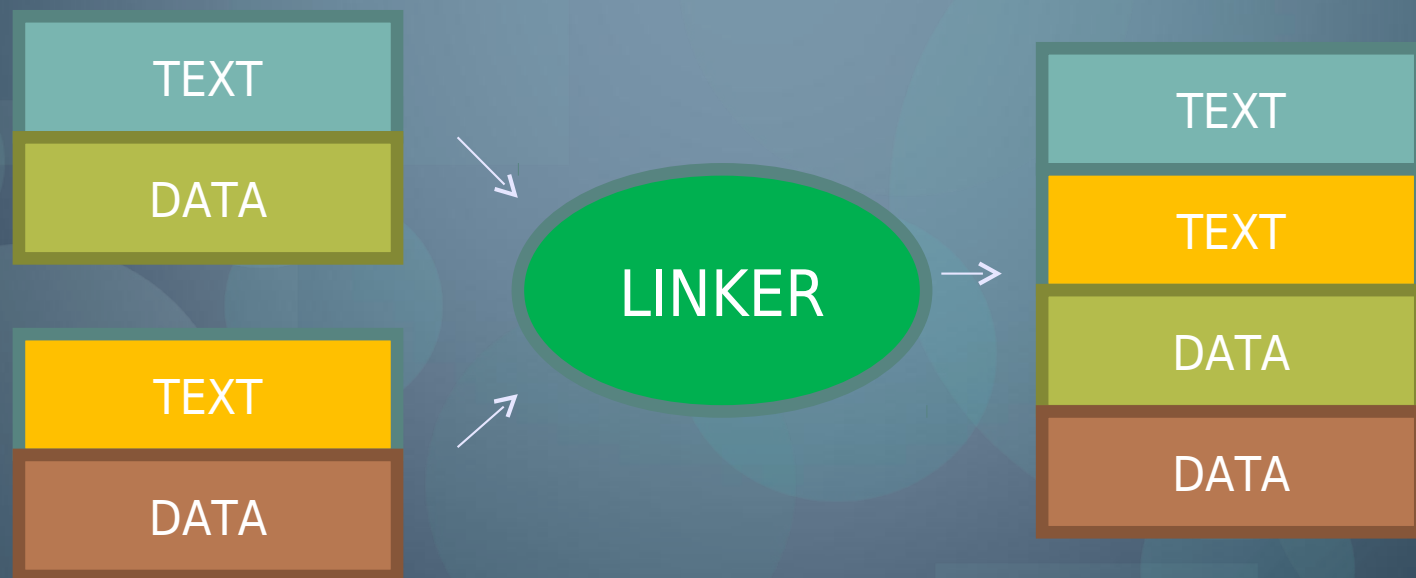
- Binutils 的核心 : BFD
- BFD: Binary File Descriptor library
- 支援多種目的檔 (PE,ELF,COFF) 和多種處理器
- 文字檔 → 抽象目的檔物件 → 最終目的檔
  - Header
  - Sections, Segment
  - Attributes
  - Data
- 其實 GNU Binutils 就是以 BFD 為基礎來實作

# Binutils – LD (GNU Linker)

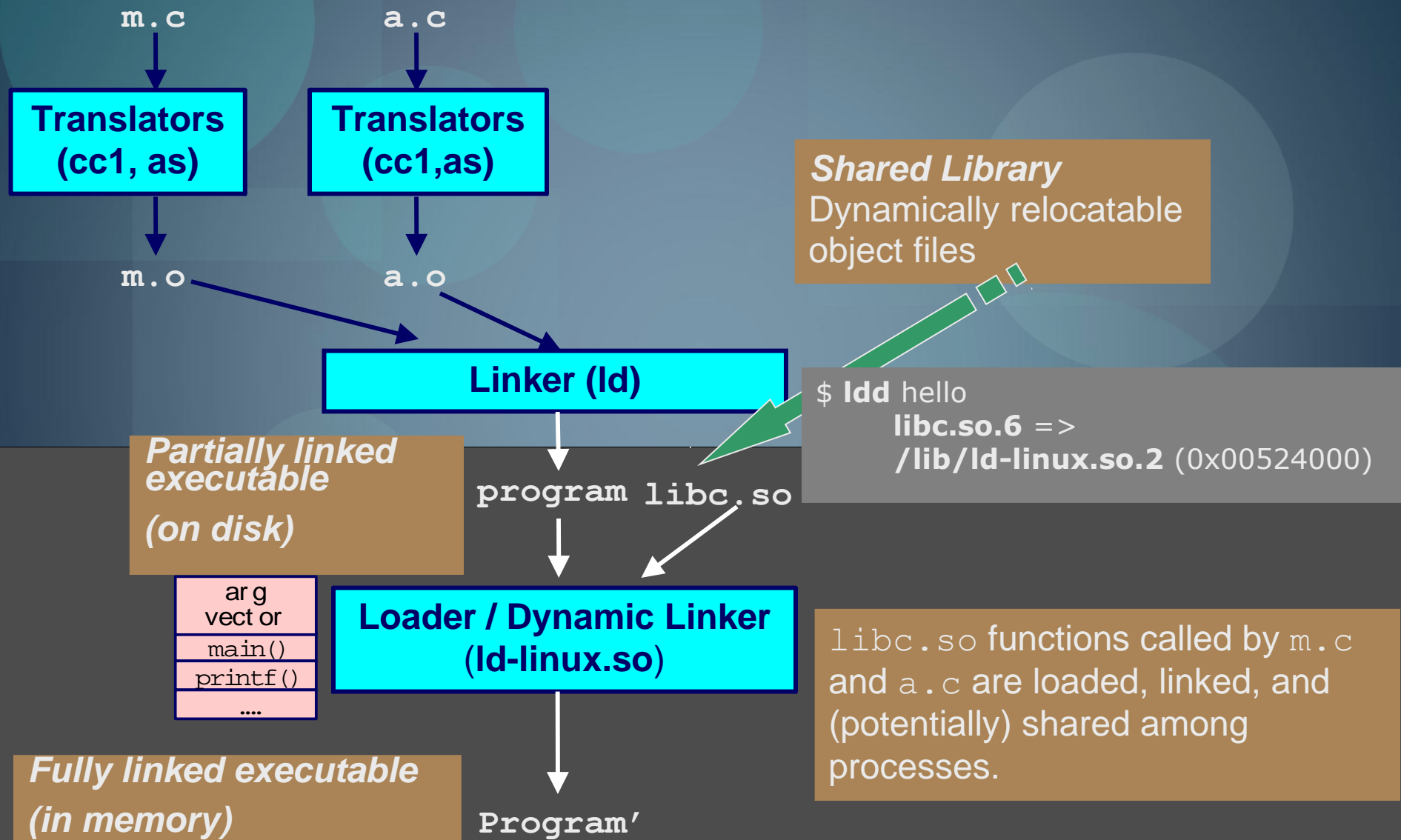
- 好久好久以前，所有的寫程式的人只能把所有的東西寫在一個檔案中
- 如果程式超過 10 萬行 (Microsoft Windows NT 4.0) 寫在一個檔案的 → 惡夢！
- 若要把程式寫在好幾個檔案中，就要有程式來進行最後彙整的動作，如果需要手動，大家是不會想要寫程式的
  - 拯救整個世界的好人，就是 Linker

# Binutils – GNU Linker

- Linker 的工作（一般靜態連結執行檔）
  - 把所有目的檔彙整成執行檔
  - 上窮碧落下黃泉 (Symbol Resolve)
  - 一切依法處理（處理 Relocation Type）

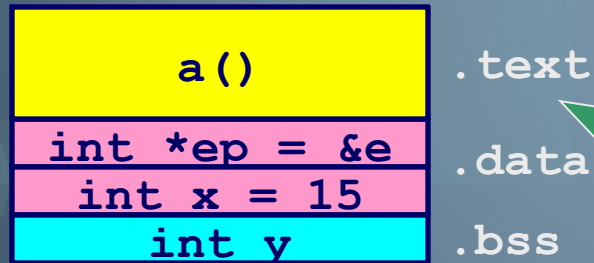
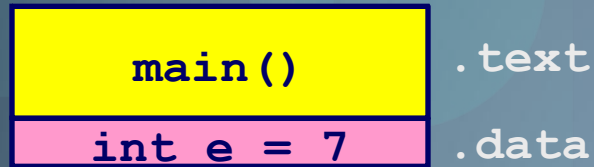


# Dynamically Linked Shared Libraries





## Relocatable Object Files



m.o

a.o

m.c

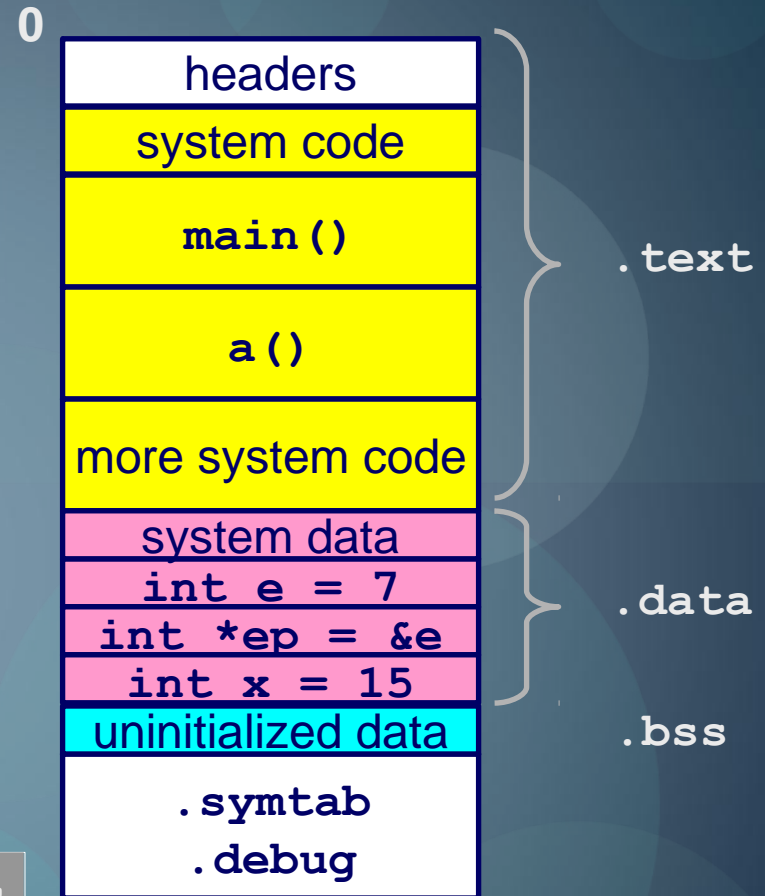
```
int e=7;

int main() {
 int r = a();
 exit(0);
}
```

```
extern int e;
int *ep=&e, x=15, y;

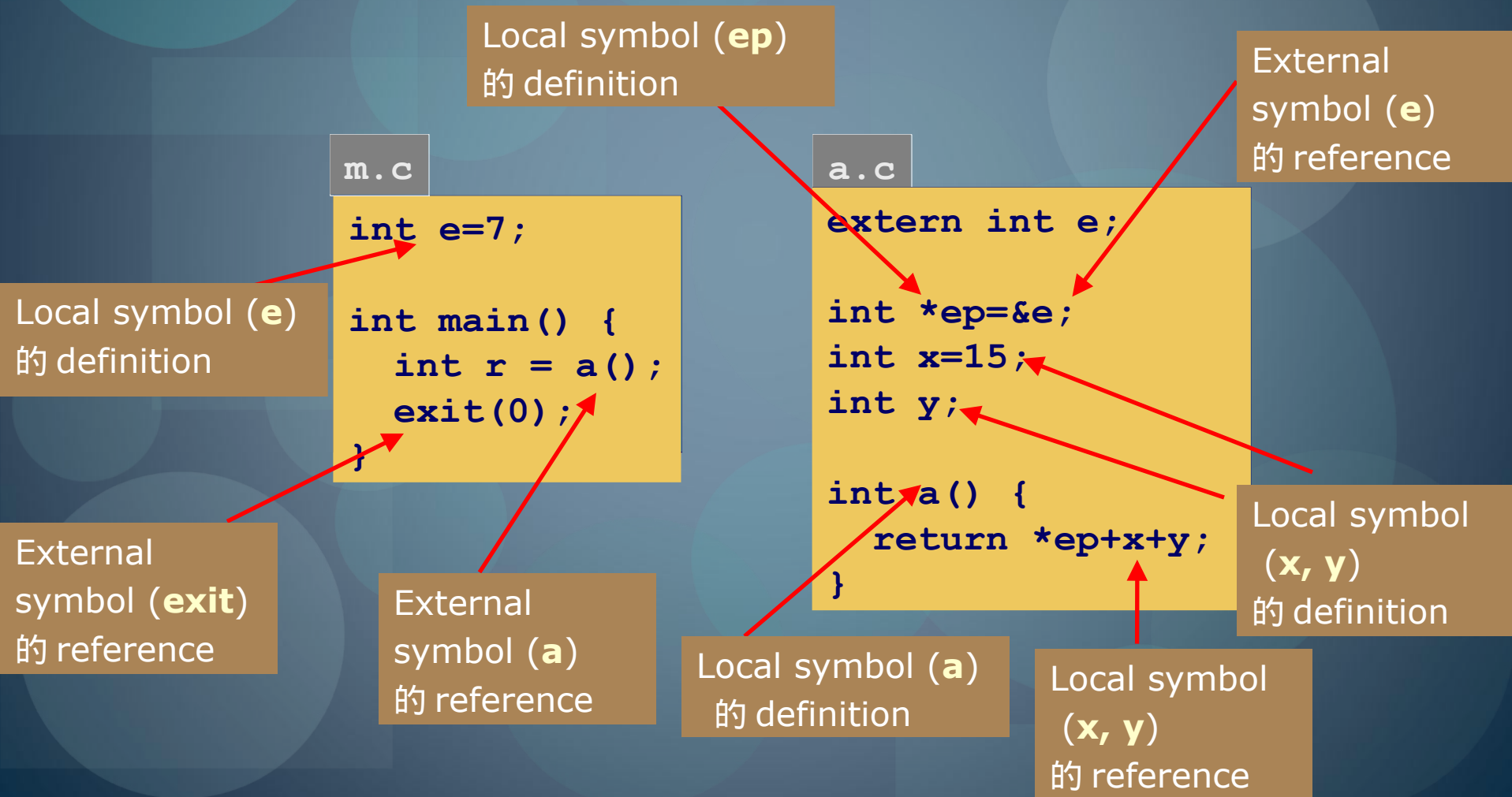
int a() {
 return *ep+x+y;
}
```

## Executable Object File



a.c

- ◆ 每個 symbol 都賦予一個特定值，一般來說就是 memory address
- ◆ Code → symbol definitions / reference
- ◆ Reference → local / external



# GCC Linker - ld

m.c

```
int e=7;

int main() {
 int r = a();
 exit(0);
}
```

Relocation Info

Disassembly of section .text:

```
00000000 <main>: 00000000 <main>:
0: 55 pushl %ebp
1: 89 e5 movl %esp,%ebp
3: e8 fc ff ff ff call 4 <main+0x4>
4: R_386_PC32 a
8: 6a 00 pushl $0x0
a: e8 fc ff ff ff call b <main+0xb>
b: R_386_PC32 exit
f: 90 nop
```

Disassembly of section .data:

```
00000000 <e>:
0: 07 00 00 00
```

```
int e=7;
```

```
int main()
{
 int r = a();
 exit(0);
}
```

```
08048530 <main>:
 8048530: 55 pushl %ebp
 8048531: 89 e5 movl %esp, %ebp
 8048533: e8 08 00 00 00 call 8048540 <a>
 8048538: 6a 00 pushl $0x0
 804853a: e8 35 ff ff ff call 8048474 <_init+0x94>
 804853f: 90
```

### Executable After Relocation and External Reference Resolution

```
08048540 <a>:
 8048540: 55 pushl %ebp
 8048541: 8b 15 1c a0 04 movl 0x804a01c, %edx
 8048546: 08
 8048547: a1 20 a0 04 08 movl 0x804a020, %eax
 804854c: 89 e5 movl %esp, %ebp
 804854e: 03 02 addl (%edx), %eax
 8048550: ec movl %ebp, %esp
 8048551: 05 d0 a3 04 addl 0x804a3d0, %eax
```

### Disassembly of section .data:

```
0804a018 <e>:
 804a018: 07 00 00 00
```

```
0804a01c <ep>:
 804a01c: 18 a0 04 08
```

```
0804a020 <x>:
 804a020: 0f 00 00 00
```

```
extern int e;
```

```
int *ep=&e;
int x=15;
int y;
```

```
int a() {
 return *ep+x+y;
}
```

# 那些 Linker 要做的事

- Linker 知道什麼：
  - 每個 .text 與 .data 區段的長度
  - .text 與 .data 區段的順序
- Linker 的運算：
  - absolute address of each label to be jumped to (internal or external) and each piece of data being referenced

# ELF

(Executable and Linkable Format)

0

## ELF header

**Program header table**  
(required for executables)

**.text section**

**.data section**

**.bss section**

**.symtab**

**rel.txt**

**.rel.data**

**.debug**

**Section header table**  
(required for relocatables)

- ◆ Page size
- ◆ Virtual address memory segment (sections)
- ◆ Segment size

- ◆ Magic number
- ◆ type (.o / .so / exec)
- ◆ Machine
- ◆ byte order
- ◆ ...

- ◆ Initialized (static) data

code

- ◆ Un-initialized (static) data
- ◆ Block started by symbol
- ◆ **Has section header but occupies no space**

注意: .dynsym 還保留

Runtime 只需要左邊欄位  
可透過“**strip**”指令去除不需要的 section

## ELF header

**Program header table**  
(required for executables)

**.text section**

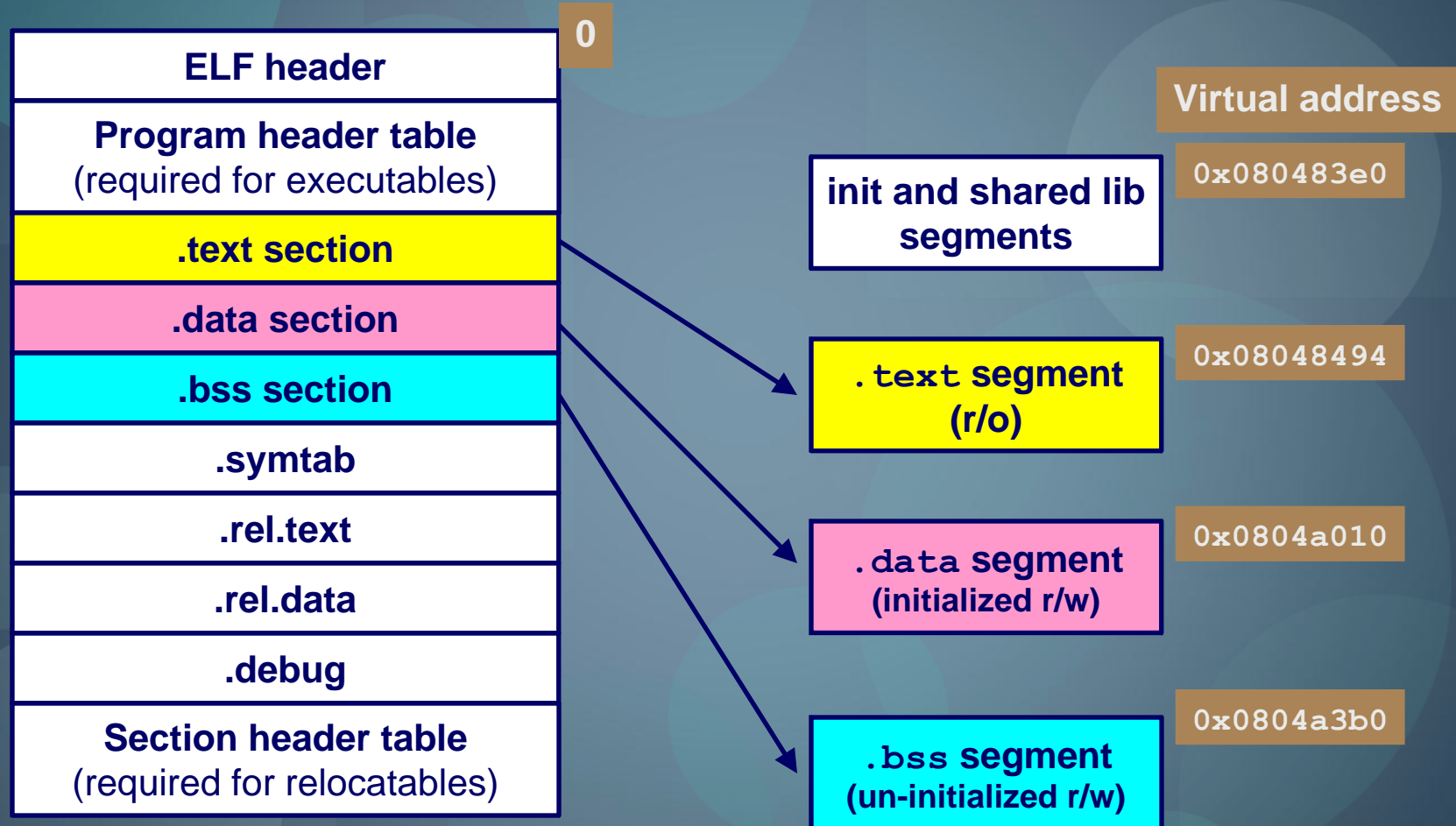
**.data section**

**.bss section**

# ELF 與 作業系統

Executable object file

Process image



## Loading ELF Binaries...



# Binutils – Gold Linker

- Gold = A New ELF Linker
  - Ian Lance Taylor (2008)
- 只專注於處理 ELF 格式
- 訴求
  - 利用 C++ 泛型程式的優點
  - 利用 多執行緒 進行連結
  - 快速連結（大型 C++ 專案）

# 連結目標程式

- 靜態連結目標檔
  - 不需要進行 Relocation
- 靜態連結執行檔
  - 需要 解決 (Resolve) 所有的 Symbol 和 Relocation
- 動態連結執行檔
  - 需要 解決 (**Resolve**) 所有的 **Symbol** 和 **Relocation**
  - Dynamic Linker 的位置
  - 需要的動態連結函示庫
  - 動態連結需要的資訊 (PLT, GOT ...etc)
- 動態連結函式庫
  - 需要 解決 (Resolve) 所有的 Symbol 和 Relocation
  - 需要的動態連結函示庫
  - 動態連結需要的資訊 (PLT, GOT ...etc)

# 動態連結器 (Dynamic Linker)

- 動態連結
  - 共用的東西不用存在好幾份
    - 執行的時候節省記憶體
  - 在電腦領域中，懶惰有時可得到很多好處
    - 用到的時候才載入
  - 版本更新的時候比較方便
    - 僅需更新特定的 Shared Library
- 在 Linux 的世界，動態連結器是 C 函式庫的一部分
  - 也就是說，每個 C 函式庫都有專屬的動態連結器
    - **GNU/Linux** → **/lib/ld-linux.so** (eglibc,glibc)
    - **Android** → **/system/bin/linker** (bionic)

# GNU/Linux 的動態連結

- 動態連結執行檔，會有連結兩次
  - Static Time Linking (ld, bfd-ld 或 gold-ld)
    - 需要解決 (Resolve) 所有的 Symbol
    - 把 External Symbol 的位址指向 stubs
      - **PLT** (procedure linkage table)
    - 沒有把真正的程式碼和資料連結進來
  - Runtime Time Linking (動態連結器：/lib/ld-linux.so)
    - 假設是使用 Lazy Binding 的狀況
    - 當 External Symbol 第一次被呼叫時，動態連結器會再解決一次 (Resolve) External Symbol
    - 動態連結器會查出真正的位址，並且放在快取中
      - **GOT** (Global Offset Table)
    - 第二次呼叫的時候，會使用快取中的位址

# PLT 與 GOT

external symbol 的位址，保存於  
**PLT** (procedure linkage table) 中

.text

....  
call

**PLT[n]** 相對應的位址

....  
.plt:

第二次呼叫時，會  
使用快取中的位址

PLT[n]:

jmp

\***GOT[n]**

PLT\_resolve[n]:

push

該項目的 Index

jmp

到 Resolver 實做

Resolver 實做:

pushl

GOT[1]

jmp

\*GOT[2]

external symbol 首次  
被呼叫時，動態連結器  
會 resolve，查出真正  
的位址，並放在快取中

.got: ...

GOT[1]: 識別動態函式庫的資訊

GOT[2]: **dl\_runtime\_resolve**

:

GOT[n]: PLT\_resolve[n]

(原本函式的位址)

# Relocation: 與平台相關的實做

- glibc elf/dynamic-link.h

```
/* This can't just be an inline function because GCC is too dumb
 to inline functions containing inlines themselves. */
define ELF_DYNAMIC_RELOCATE(map, lazy, consider_profile) \
do { \
 int edr_lazy = elf_machine_runtime_setup ((map), (lazy), \
 (consider_profile)); \
 ELF_DYNAMIC_DO_REL ((map), edr_lazy); \
 ELF_DYNAMIC_DO_RELA ((map), edr_lazy); \
} while (0)
```

- glibc sysdeps/i386/dl-machine.h

```
/* Set up the loaded object described by L so its unrelocated PLT
 entries will jump to the on-demand fixup code in dl-runtime.c.
*/
static inline int __attribute__((unused, always_inline))
elf_machine_runtime_setup (struct link_map *l, int lazy, int
profile)
{
 :
 got[1] = (Elf32_Addr) l; /* Identify this shared object. */
 :
 got[2] = (Elf32_Addr) &_dl_runtime_resolve;
 :
}
```

ELF resolver

# 動態連結的美麗與哀愁

- 位址無關程式碼 (PIC, Position-independent code)
  - 動態連結的函式在被載入時，才知道自己的位址
  - 為避免無謂的重新定址，動態連結函式庫會使用 PIC
  - 所有全域的位址都以 BASE + OFFSET 型式存在
  - 使用 PIC 的時候，效率會降低
- 延遲載入
  - 大部分的狀況是好處大於壞處
  - 可是會影響程式的效能
  - **Prelink** 技術用以解決載入時間較長的問題
    - 甚至可用於 Linux Kernel Module
    - [http://elinux.org/images/8/89/LKM\\_Preresolver\\_ELC-E\\_2010.pdf](http://elinux.org/images/8/89/LKM_Preresolver_ELC-E_2010.pdf)



# 總結

---

- 微言大義
  - Hello World 程式啟動流程
  - Hello World 程式編譯流程
- Compiler Driver
- 三大步驟
  - 五大階段，三大法門
  - 編譯，組譯，連結

# 參考資料

- Loader and Linker, John R. Levine 2000
- 程序員的自我修養 – 連結、載入與程式庫
- LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation
  - <http://llvm.org/pubs/2004-01-30-CGO-LLVM.html>
- GCC, the GNU Compiler Collection
  - <http://gcc.gnu.org/>
- GNU Binutils
  - <http://www.gnu.org/software/binutils/>
- Embedded GLIBC
  - <http://www.eglibc.org/home>