二、開發工具介紹

工欲善其事，必先利其器

Linux 程式設計工具集

- gcc (GNU Compiler Collection)
  - http://www.gnu.org/software/gcc/onlinedocs/
- make (Project Management Tool)
  - http://www.gnu.org/manual/make-3.79.1/make.html
- automake與autoconfig
  - http://www.gnu.org/manual/autoconf-2.57/autoconf.html
  - http://www.gnu.org/manual/automake-1.7.2/automake.html
- gdb (GNU debugger)
  - http://www.gnu.org/manual/gdb-5.1.1/gdb.html

GNU Compiler Collection

GCC簡介

GCC

- Compiles C, C++, Objective C, Fortran, Pascal, Modula-3, Ada, etc..
  - gcc compiles with "C"
  - g++ compiles with "C++"

GCC執行範例

CODE. 2-1

$gcc hello.c –o hello

$./hello

Hello world!

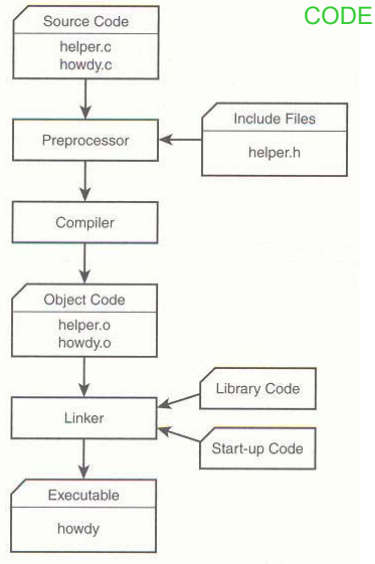
```
#include <stdio.h>

int main(void){
        printf("Hello world!\n");
        return 0;
}
```


GCC

- 四個步驟
  - 前置處理(Preprocessing)
    - gcc –E hello.c –o hello.cpp
  - 編譯(Compilation)
    - gcc –x cpp-output –c hello.cpp –o hello.o
  - 組譯(Assembly)
  - 連結(Linking)
    - gcc hello.o –o hello

CODE. 2-2



GCC檔案副檔名與檔案類型的對應

.c	C原始檔
.C .cc	C++原始檔
.i	前置處理後的C原始檔
.ii	前置處理後的C++原始檔
.S .s	組合語言原始檔
.o	目標檔(compiled object code)
.a	靜態函式庫
.so	動態函式庫

GCC命列列參數(1/3)

CODE. 2-3

- -o FILE
  - 指定輸出檔名
- -c
  - 只編譯而不連結
- -DFOO=BAR
  - 定義preprocessor的FOO變數等於BAR
- -IDIRNAMEs
  - 讓gcc找尋DIRNAMEs目錄下的header file
- -LDIRNAMEs
  - 讓gcc找尋DIRNAMEs目錄下的library

GCC命列列參數(2/3)

CODE. 2-4

- -static
  - 連結靜態函式庫，預設是連結動態函式庫
- -lFOO
  - 連結到libFOO
- -ansi
  - 支援ANSI/ISO C標準，並將GNU extensions與ANSI/ISO標準衝突的部分關閉
- -pedantic
  - 顯示所有ANSI/ISO的warning訊息
- -pedantic-errors
  - 顯示所有ANSI/ISO的error訊息

GCC命列列參數(3/3)

CODE. 2-5

- -traditional
  - 使用K&R C語言語法
- -w
  - 取消所有warning訊息
- -Wall
  - 要求GCC顯示所有的警告訊息
- -werror
  - 將所有的warning轉成error，如此一來，warning就會讓編譯過程停止
- -MM
  - 產生make-compatible dependency list
- -v
  - 顯示所有編譯過程

GCC警告與錯誤訊息選項(1/2)

- -Wcomment
  - 如果使用nested /* 則產生warning
- -Wformat
  - 如果傳遞到printf的參數型別與format不符
- -Wmain
  - 如果main的傳回值不是int或是被呼叫的參數不對
- -Wparentheses
  - 刮號用在指定(assign)上，例如(a=5)
- -Wswitch
  - switch中的case比可能的值還少時(只適用在enum型別)
- -Wunused
  - 變數宣告但是沒有使用，或是static function已經宣告但是沒有定義

GCC警告與錯誤訊息選項(2/2)

CODE. 2-6

- -Wuninitialized
  - 變數未初始化就開始用
- -Winline
  - 函式無法inlined
- -Wmissing-declarations
  - 函式已經被定義，但是沒有在任何檔頭宣告
- -Wlong-long
  - 使用long long
- -Werror
  - 將所有的warning變成error

GCC最佳化選項(1/3)

- -ffloat-store
  - 阻止浮點數變數佔用CPU的暫存器
- -ffast-math
  - 產生浮點數最佳化，如此會較快但是會違反IEEE和ANSI/ISO標準
- -finline-functions
  - 把所有簡單的函式變成inline function
- -funroll-loops
  - 把所有已知固定次數的loop展開

GCC最佳化選項(2/3)

- -fomit-frame-pointer
  - 取消不必要儲存在CPU暫存器的的frame pointer
- -fschedule-insns
  - 將會造成CPU stall的instruction順序重排
- -fschedule-insns2
  - 第二回合，與-fschedule-insns相似
- -fmove-all-movables
  - 將迴圈中固定不變的運算搬到迴圈外面

GCC最佳化選項(3/3)

- -O
  - 與-O1相同
- -O1
  - 根據target processor來做最佳化
  - Thread jumps optimizations(減少jump的次數)
  - Deferred stack pops(減少從stack pop的次數)
- -O2
  - 包含-O1的所有最佳化
  - 檢查是否有指令會造成CPU stall，這項最佳化與CPU的架構十分相關
- -O3
  - 包含-O2的所有最佳化
  - 把loop展開
  - 使用其它CPU的特性

GCC最佳化比較(1/2)

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

int main(void)
{
    double pi = M_PI; /* Defined in <math.h> */
    double pisqrt;
    long i;

    for(i = 0; i < 10000000; ++i) {
        pisqrt = sqrt(pi);
    }
    return 0;
}
```

GCC最佳化比較(2/2)

Flag/Optimization	*Average Execution Time*
<none>	5.43 seconds
-O1	2.74 seconds
-O2	2.83 seconds
-O3	2.76 seconds
-ffloat-store	5.41 seconds
-ffast-math	5.46 seconds
-funroll-loops	5.44 seconds
-fschedule-insns	5.45 seconds
-fschedule-insns2	5.44 seconds

GCC除錯選項(1/2)

- -g
  - 包含除錯資訊到輸出檔(標準除錯資訊)，與-g2相同
- -g1
  - 只產生backtrace與stack dump所需要的資訊
- -g2
  - 包含符號表，行數，區域和全域變數的資訊
- -g3
  - 包含-g2的所有資訊，加上原始程式碼裡macro
- -ggdb
  - 包含gdb的除錯資訊到輸出檔(只有gdb看的懂)

GCC除錯選項(2/2)

- -pg
  - 加入profiling資訊，日後可以使用gprof程式讀取
  - 可用來找尋程式瓶頸
- -save-temps
  - 保留中間(intermediate)檔案
- -Q
  - 顯示編譯每個步驟的時間

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
```


GCC硬體架構選項

- -mcpu=CPU TYPE
  - 選擇CPU instruction schedule，可以為i386, i486, i586, pentium, i686和pentiumpro
- -m386
  - 相等於-mcpu=i386
- -m486
  - 相等於-mcpu=i486
- -mpentium
  - 相等於-mcpu=pentium
- -mpentiumpro
  - 相等於-mcpu=pentiumpro
- -march=CPU TYPE
  - 選擇使用何種指令集，如果已經用-mcpu指定，還是要用-march指定指令集

如何使用程式庫

- $gcc myapp.c –I /home/native/include
- $gcc myapp.c –L /home/native/lib –lnew
- $gcc myapp.c –I /home/native/include –L /home/native/lin –lnew –o myapp
- $gcc cursesapp.c –lncurses –static

GNU C Extensions

CODE. 2-7

- long long
- inline functions
- function and variable attributes
- case ranges

Project Management Tool

Make簡介

Make

- 為什麼需要make?
  - 一個project可能包含許多的檔案
  - 編譯的過程可能需要許多的指令與參數
- make可以用來解決上述問題
  - make會依據Makefile的內容依照順序執行一連串的命令
  - 另外make會比對檔案的時間戳記(time stamp)來決定是否需要重建(rebuild)，以加快速度

Makefile

- 檔名可以為GNUmakefile, makefile或Makefile
- Makefile的語法如下：

target: dependency [dependency [...]]
    command
    command
    [...]

PS. command之前為一個以上的tab字元

Makefile 範例

- $make helper.o
- $make howdy.o
- $make helper.o howdy.o

```
howdy: howdy.o helper.o helper.h
	gcc howdy.o helper.o -o
howdy

helper.o: helper.c helper.h
	gcc -c helper.c

howdy.o: howdy.c
	gcc -c howdy.c

hello: hello.c
	gcc hello.c -o hello

all: howdy hello

clean:
	rm howdy hello *.o
```

Makefile

- 第一個target為預設的target
- 預設的target可以執行其它的target來建立自己所需的dependencies
- 沒有dependency的target稱為Phony target
  - Phony target不會自動執行
  - Phony target必須由make直接指定phony target 來執行，例如make clean

Makefile 變數

- Makefile中可以使用變數，以方便未來修改
- 變數宣告

  VARNAME = <some-text>
- 變數使用

  $(VARNAME)

Automatic Variable

- $@
  - rule target的名稱
- $<
  - 第一個dependency的名稱
- $^
  - 所有dependencies列表，以空格相隔
- $?
  - 所有比target還新的dependencies列表，以空格相隔
- $(@D)
  - Target名稱的目錄部分(如果target是在子目錄下)
- $(@F)
  - Target名稱的檔案部分(如果target是在子目錄下)

Predefined Variables

- AR Archive-maintenance programs
- AS Program to do assembly
- CC Program for compiling C programs
- CPP C Preprocessor program
- RM Program to remove files
- ARFLAGS Flags for the AR program
- ASFLAGS Flags for the AS program
- CFLASGS Flags for the CC program
- CPPFLAGS Flags for the CPP program
- LDFLAGS Flags for the linker (ld)

Implicit Rules and Pattern Rules

- 在Makefile中指定的rule稱為explicit rule，這些explicit rule可能自動衍生出一些implicit rules
- Pattern rules

```
%.o: %.c
    $(CC) -c $< -o $@
```


常用的makefile targets名稱

- all
- dist
  - 建立包裝檔
- clean
  - 把產生的檔案刪除
- install
  - 安裝檔案
- test
  - 測試程式是否有問題

automake & autoconf

程式的可移植性(Portability)

- 程式碼在不修改的狀況下可以在不同的平台上編譯執行
- 軟體可能的問題
  - 編譯器種類不同
  - 缺少函式庫
  - 系統服務運作方式不同
  - 檔案系統不同
- 硬體可能的問題
  - Big-endian與little-endian的轉換

如何增強程式的可移植性

- 避免使用特定系統才有的函式
- 把使用到特定系統的程式碼獨立出來
- 儘量使用已經存在的介面，而不要自己重寫一個
- 使用標準的介面，像是POSIX

autoconf 與automake介紹

- 檢查系統相容性
  - 檢查library是否存在
  - 檢查header file是否存在
  - 檢查program是否存在
- 透過這些檢查，會自動定義一些macro，讓編譯器知道目前系統的情況，因此程式中就可以用#ifdef XXX 來避免不相容的問題

configure.in

- 可以直接編寫或是執行autoscan產生，檔案結構如下：

```
AC_INIT(test.c)

dnl Checks for programs.

dnl Checks for libraries.

dnl Checks for header files.

dnl Checks for typedefs, structures, and compiler characteristics.

dnl Checks for library functions.

AC_OUTPUT()
```

- ifnames *.c 列出所有conditional macro，檢查configure.in是否有遺漏處理這些 macro

configure.in

- AC_INIT(unique_file_in_source_dir)
  - 用該檔案來判斷目前是否在source的目錄中，如果目前目錄包含該檔按即代表是
- AC_OUTPUT([file…[,extra_cmds[,init_cmds]]])
  - file 代表產生的檔案，以空白隔開
  - 每一個file都是由file.in產生的，例如Makefile是由Makefile.in產生
  - extra_cmds 會加入config.status後面
  - init_cmds會在extra_cmds之前加入config.status

Test for alternative programs

- AC_PROG_AWK
  - 依照順序找尋mawk, gawk, nawk或awk
- AC_PROG_CC
  - 決定使用那個C編譯器(設定變數CC)
- AC_PROG_CC_C_O
  - 判斷編譯器是否可用-c或-o
- AC_PROG_CPP
  - 設定變數CPP
- AC_PROG_INSTALL
  - 設定變數INSTALL成BSD-compatible的install程式

Test for alternative programs

- AC_PROG_LEX
  - 依照順序找尋flex,lex(設定變數LEX)
- AC_PROG_LN_S
  - 如果系統支援symbolic link則將變數LN_S設定成ln –s，否則設定成ln
- AC_PROG_RANLIB
  - 如果存在ranlib程式則將變數RANLIB設成它
- AC_PROG_YACC
  - 依照順序找尋bison, byacc或yacc(設定變數YACC)

Tests for library functions

- AC_CHECK_LIB(lib, function)
  - 測試function是否在lib中，如果成功的話將-llib加入變數LIB中
- AC_FUNC_GETLOADAVG
  - 如果系統支援getloadavg的函式，將所需的libarary加入變數LIB中
- AC_FUNC_GETPGRP
  - 測試getprgrp函式是否需要參數
- AC_FUNC_MEMCMP
  - 如果memcmp不存在，將memcp.o加入變數LIBOBJS
- AC_FUNC_MMAP
  - 如果mmap存在則設定HAVE_MMAP

Tests for header files

- AC_DECL_SYS_SIGLIST
  - 如果signal.h或unistd.h定義sys_siglist，則定義SYS_SIGLIST_DECLARED
- AC_HEADER_DIRENT
  - 依序搜尋dirent.h (HAVE_DIRENT_H), sysdir/ndir.h (HAVE_SYS_NDIR_H), sys/dir.h (HAVE_SYS_DIR_H), ndir.h (HAVE_NDIR_H)看哪一個檔案有定義DIR
- AC_HEADER_STDC
  - 如果系統有ANSI/ISO C的相容檔頭則定義STDC_HEADERS
- AC_HEADER_SYS_WAIT
  - 如果系統有與POSIX相容的sys/wait.h則定義HAVE_SYS_WAIT

Tests for structures

- AC_HEADER_TIME
  - 如果time.h與sys/time.h都存在，設定變數TIME_WITH_SYS_TIME
- AC_STRUCT_ST_BLKSIZE
  - 如果struct stat包含st_blksize這個成員，則定義HAVE_ST_BLKSIZE
- AC_STRUCT_ST_BLOCKS
  - 如果struct stat包含st_blocks這個成員，則定義HAVE_ST_BLOCKS
- AC_STRUCT_TIMEZONE
  - 指出如何取得timezone，如果定義HAVE_TM_ZONE代表struct tm包含tm_zone這個成員，如果定義HAVE_TZNAME代表找到tzname的陣列

Tests for typedefs

- AC_TYPE_GETGROUPS
  - 根據getgroups所需的參數，設定GETGROUPS_T為gid_t或int
- AC_TYPE_MODE_T
  - 如果mode_t沒有定義則定義mode_t為int
- AC_TYPE_PID_T
  - 如果pid_t沒有定義則定義pid_t為int
- AC_TYPE_SIZE_T
  - 如果size_t沒有定義則定義size_t為unsigned
- AC_TYPE_UID_T
  - 如果uid_t沒有定義則定義uid_t與gid_t為int

Tests of compiler behavior

- AC_C_BIGENDIAN
  - 如果是bigendian則定義WORDS_BIGENDIAN
- AC_C_CONST
  - 如果compiler不支援const宣告，把const定義成空白
- AC_C_INLINE
  - 如果compiler不支援inline，則把inline,__inlinde__或__inline定義成空白
- AC_C_CHAR_UNSIGNED
  - 如果char為unsigned，定義CHAR_UNSIGNED
- AC_C_LONG_DOUBLE
  - 如果compiler支援long double資料型態則定義HAVE_LONG_DOUBLE

Generic Macros

- AC_TRY_CPP(includes)
  - 讓preprocessor去include，看是否有錯誤發生
- AC_EGREP_HEADER(pattern, header, action_if_found)
  - 尋找header檔中是否有出現pattern，如果有則執行action_if_found的script
- AC_TRY_COMPLETE(includes, function_body)
  - 用來測試編譯器的語法
- AC_TRY_LINK(includes, function_body)
  - 把AC_TRY_COMPLETE加入連結(link)測試
- AC_TRY_RUN(program)
  - 測試程式編譯連結後是否能執行
- AC_CHECK_PROG
  - 測試程式是否存在

Makefile.am

- 給automake產生Makefile.in用
- 其內容包含
  - AUTOMAKE_OPTIONS=foreign
    - 記錄嚴謹度，主要是看套件是否符合GNU標準
  - bin_PROGRAMS=test
    - 所要產生的執行檔檔名
  - test_SOURCES=test.c test.h
    - test的dependencies
- 使用automake --add-missing產生Makeifle.in

autoconf與automake的製作流程

1. 執行autoscan 產生configure.scan
2. 將configure.scan改名為configure.in並編輯其內容
3. aclocal 產生aclocal.m4
4. autoconf 產生configure
5. 建立Makefile.am
6. 執行automake --add-missing產生Makefile.in

Debugger

GDB簡介

gdb 除錯器

- GNU所提供的除錯器
- 用gcc compile時必須加入除錯的資訊才可以
  - gcc –g test.c –o test
  - gcc –ggdb test.c –o test
- 最好不要與-O的最佳化選項一起使用
  - 最佳化後程式的執行流程可能改變
  - 最佳化後有些變數可能消失
- 含有除錯資訊的binary可以用strip指令將之消除

啟動gdb

- gdb progname [corefile]
  - $gdb a.out
  - $gdb a.out core
- gdb command-line option
  - -q 不要顯示版權畫面
  - -d dirname 告知gdb在哪裡可以找到source code

gdb常用指令(1/4)

- help
  - 查詢指令
- set args
  - 設定argv[]的值
- run
  - 從程式的開頭處開始執行
- backtrace
  - 顯示call stack
- list
  - 顯示原始程式碼

gdb常用指令(2/4)

- print
  - 顯示變數的值
  - print i
  - print argv@5
  - print argv[0]@5
- whatis
  - 顯示變數的型別

gdb常用指令(3/4)

- break
  - 設定中斷點
  - break linenum
  - break funcname
  - break filename:linenum
  - break filename:funcname
  - break xxx if expr
- info breakpoints
  - 顯示目前設定的中斷點資料
- disable
  - 將中斷點關閉
- enable
  - 將中斷點開啟
- delete
  - 刪除中斷點
- continue
  - 遇到中斷點後繼續執行

gdb常用指令(4/4)

- ptype type
  - 顯示structure的定義
- set variable varname=value
  - 設定變數的值
- next
  - 一次執行一整個function
- step
  - 一次執行一個指令
- call name(args)
  - 呼叫函式
- finsh
  - 跑完目前函式，並印出return value
- return value
  - 終止目前函式的執行，直接回傳value

gdb進階指令(1/3)

- variable scope and context
  - 在目前context中可用的變數為active，不可用的變數為inactive
    - 當某個函式在執行時，該函數的區域變數為active，當該函數結束時，這些區域變數變為inactive
    - 全域變數永遠都為active
  - print 'foo.c'::baz (foo.c為檔案)
  - print &test::idx (test為函式)

gdb進階指令(2/3)

- Traversing the call stack
  - gdb提供兩個指令在call stack中移動
  - where
    - 顯示目前的call stack
  - up
    - 往上一層
  - down
    - 往下一層

gdb進階指令(3/3)

- Attaching to a running program
  - $gdb program pid
  - 或是在gdb中打下面兩個指令
    - file program
    - attach pid

Library

函式庫簡介

函式庫(library)

- 使用函式庫有下列優點
  - 讓程式碼可以重複使用
  - 除錯所花的時間較少
- 函式庫可以分成下列兩類
  - static library
    在編譯其間就將函式庫連結到執行檔中，其優點是執行速度較快，但是浪費系統資源。此外如果函式庫發現bug，必須將所有使用該函式庫的程式重新編譯
  - shared library
    在執行其間將才將函式庫載入記憶體，其優點是節省記憶體，缺點是執行速度較慢。當函式庫發現bug，只需要重新編譯函式庫即可

製作與使用靜態函式庫

- 基本上靜態函式庫是一群object files的集合，因此製作方式為先用編譯器(例如gcc)編譯出object file，然後在用ar指令將這些object files包成一個檔案(副檔名為 .a)

print.c
```
void test(){
    printf("library test\n");
}
```

print2.c
```
void test(){
    printf("library test2\n");
}
```

main.c
```
int main(void){
    test();
    test2();
}
```

```
gcc -c print.c                    <=建立object file
gcc -c print2.c                   <=建立object file
ar -rcs libprint.a print.o print2.o   <=建立static library
gcc -o main main.c -lprint -L .   <=使用library
```


動態函式庫

- 動態函式庫是在執行時期才載入，因此平常動態函式庫檔案是散佈在檔案系統中，例如/lib，/usr/lib，/usr/local/lib(這三個)等等。這些散佈的路徑記錄在/etc/ld.so.conf中
- 為了加速找尋動態函式庫的速度，ld.so(動態函式庫連結器)會根據/etc/ld.so.cache的內容來查詢，因此當/etc/ld.so.conf裡記錄的函式庫有更動的話，必須執行ldconfig指令來更新/etc/ld.so.cache的內容
- ld.so除了找尋/etc/ld.so.cache之外，也會根據使用者的環境變數LD_LIBRARY_PATH來找尋，因此一般使用者可以藉由設定LD_LIBRARY_PATH的環境變數來使用放在其它位置(例如自己的home目錄下)的shared library
- 同一個動態函式庫可能有不同的版本，他的命名方式通常是 libxxx-major.minor.patch.so，例如libc-2.2.93.so

製作與使用動態函式庫

- 製作與使用動態函式庫

```
gcc -fPIC print.c
gcc -fPIC print2.c
gcc -shared -Wl,-soname,libprint.so.2 -o libprint-2.0.0.so print.o print2.o –lc
ln -s libprint-2.0.0.so libprint.so.2      <=給ld.so用，也就是run time
ln -s libprint.so.2 libprint.so                    <=給ld用，也就是compile time
gcc -o main main.c -lprint -L .
```

- 執行
  - export LD_LIBRARY_PATH=.
  - ./main

動態載入動態函式庫(1/2)

```
#include <dlfcn.h>

    void *dlopen(const char *filename, int flag);
    const char *dlerror(void);
    void *dlsym(void *handle, char *symbol);
    int dlclose(void *handle);
```

- 透過此頁函式，在編譯時期完全不需要動態函式庫的存在，直到執行時期才需要
- dlopen函式用來載入動態函式庫，第一個參數為動態函式庫檔名，如果其不是絕對路徑，會根據下列順序來找尋
  - LD_LIBRARY_PATH
  - /etc/ld.so.cache
  - /lib
  - /usr/lib
- dlopen函式的第二個參數用來指定哪時候會去解析(resolve)外在的參考(函式庫裡還有用到其它函式庫)，可以有下列值
  - RTLD_LAZY: 當呼叫才做解析的動作
  - RTLD_NOW: 當dlopen時就做解析的動作
  - RTLD_GLOBAL: 將所有的全域變數export出來
- dlsym函式用來取得某個符號或函式的指標
- dlclose函式用來關閉不用的動態函式庫

動態載入動態函式庫(2/2)

```
#include <dlfcn.h>

int main(void){
        void *handle;
        void (*func1)();
        void (*func2)();
        int *global;

        handle=dlopen("./libprint.so",RTLD_NOW);          //載入libprint.so動態函式庫

        dlerror();                                        //清除之前的錯誤訊息

        func1=dlsym(handle,"test");                       //取得test函式指標
        func2=dlsym(handle,"test2");                      //取得test2函式指標
        global=dlsym(handle,"global");                    //取得global變數指標

        func1();                                          //呼叫func1，也就是test( )
        func2();                                          //呼叫func2，也就是test2( )
        printf("global=%d\n",*global);                    //將global變數的值印出來

        dlclose(handle);                                  //關閉動態函式庫
}
```