

SIGSEGV時在做什麼？ 有沒有**GDB**？可以來除錯嗎？

講師：Yuto

\$Outline

- ▶ 00 環境準備
- ▶ 01 程式編譯與執行
- ▶ 02 使用**GDB**除錯
- ▶ 03 程式流程與斷點
- ▶ 04 **gdbinit**與插件
- ▶ 05 變數檢視
- ▶ 06 程式競賽的應用
- ▶ 06 函式呼叫
- ▶ 07 記憶體傾印

\$whoami

- ▶ 羅崧瑋/Yuto
- ▶ 資工系大三
- ▶ 技能點：
 - ▶ Linux、逆向工程、pwn
 - ▶ 網頁前後端

\$先備知識

- ▶ 基本撰寫 C 語言的能力
 - ▶ 大概知道指標的 *、& 的作用
- ▶ Windows/Unix-like Shell 的基本操作
- ▶ 擁有學習熱忱的心

\$為什麼該接觸 **GNU/Linux** 開發工具

成功大學 黃敬群 教授

- ▶ 學習和研究
- ▶ 專業視野
- ▶ 工作機會
- ▶ 實作導向

[source](#)

\$為什麼該接觸 **GNU/Linux** 開發工具

- ▶ 學習和研究
 - ▶ 提供了很多高品質的開源工具
 - ▶ 豐富的公開文件以及教學
 - ▶ 可以透過閱讀原始碼，站在巨人的肩膀
- ▶ 專業視野
- ▶ 工作機會
- ▶ 實作導向

[source](#)

\$為什麼該接觸 **GNU/Linux** 開發工具

成功大學 黃敬群 教授

- ▶ 學習和研究
- ▶ 專業視野
 - ▶ 透過這些專案認識世界各地的天才
 - ▶ 甚至可以獲得工作機會([LFX Mentorship](#))
- ▶ 工作機會
- ▶ 實作導向

[source](#)

\$為什麼該接觸 **GNU/Linux** 開發工具

成功大學 黃敬群 教授

- ▶ 學習和研究
- ▶ 專業視野
- ▶ 工作機會
 - ▶ 人類共享、自由、開放的理念
 - ▶ 會比同齡人有更多的實務經驗
- ▶ 實作導向

[source](#)

\$為什麼該接觸 **GNU/Linux** 開發工具

成功大學 黃敬群 教授

- ▶ 學習和研究
- ▶ 專業視野
- ▶ 工作機會
- ▶ 實作導向
 - ▶ 解決真實世界的問題

[source](#)

這些我都不會

窩該怎麼辦



加入科學開源服務社！！

加入科學用服務社！！
不要亂置入

- ▶ 開闊視野
- ▶ 缺什麼，補什麼

00 環境準備

學習最難的莫過於開始了

\$ 實驗環境

- ▶ Editor : vscode(建議)、vim...
- ▶ Compiler : gcc
- ▶ Debugger : gdb

\$ 實驗環境-Ubuntu

- ▶ sudo apt install vim gcc

\$ 實驗環境 - Windows

- ▶ 我熱愛 Linux
 - ▶ 用 WSL 裝 Ubuntu
 - ▶ 回上一步
- ▶ 我還是要用 Windows
 - ▶ Chocolatey
 - ▶ MinGW-w64

\$ 實驗環境 - 簡報 & Labs

- ▶ git clone https://github.com/yuto0226/gdb_tutorial.git

01 程式碼編譯與執行

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    char str[] = "Hello, world!\n";      // 宣告字串 str
    printf(str);                      // 用 printf 印出字串 str

    return 0;
}
```

Celinglass



老師我的程式碼怎麼編譯那麼久？

Softpedia.cpp - Code::Blocks 16.01

File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help

<global>

Management

Projects Symbols Files

Workspace

Softpedia

Sources

main.cpp

Softpedia.cpp

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     int a;
8     int b;
9     cin>> a>> b;
10    int sum = a+b;
11    cout << sum << endl;
12    return 0;
13 }
14
```

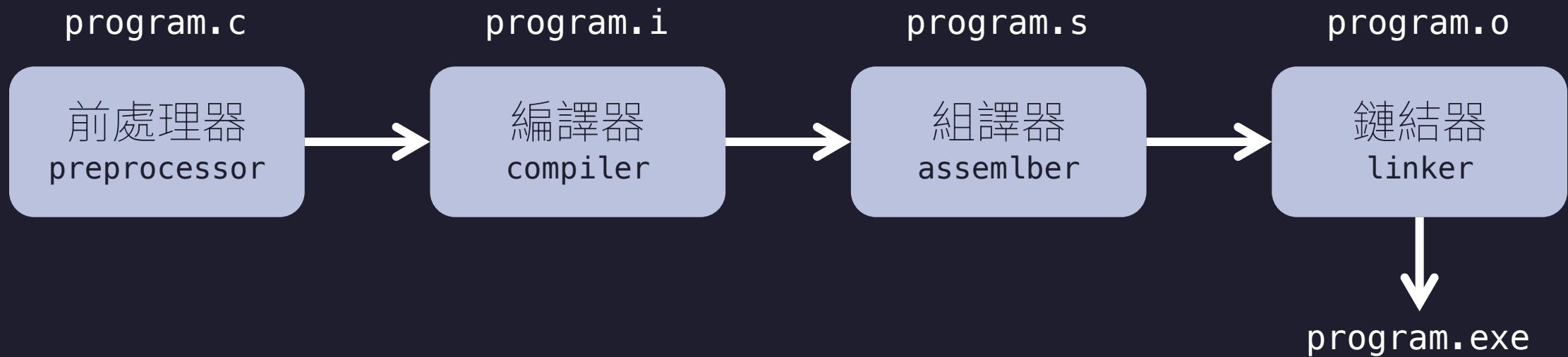
Logs & others

Code::Blocks

NativeParser::CreateParser(): Finish creating a new parser for project 'Softpedia'
NativeParser::OnParserEnd(): Project 'Softpedia' parsing stage done!
C:\SoftpediaTest\Softpedia C++ test\Softpedia\Softpedia.cpp
NativeParser::CreateParser(): Finish creating a new parser for project '"NONE"'
Switch parser to project '"NONE"'
NativeParser::OnParserEnd(): Project '"NONE"' parsing stage done!

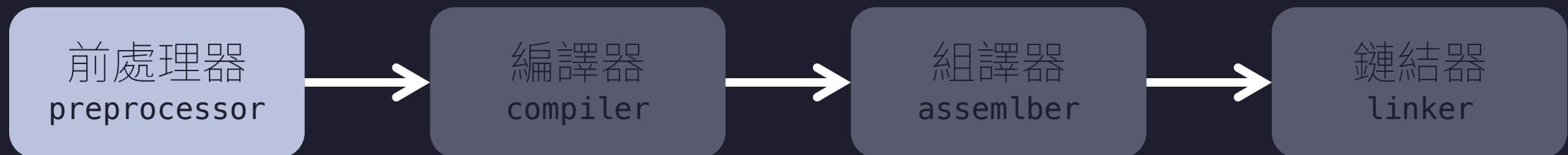
C:\SoftpediaTest\Softpedia C++ test\Softpedia Windows (CR+LF) | WINDOWS-1252 | Line 1, Column 20 | Insert | Read/Write | default

\$編譯程式



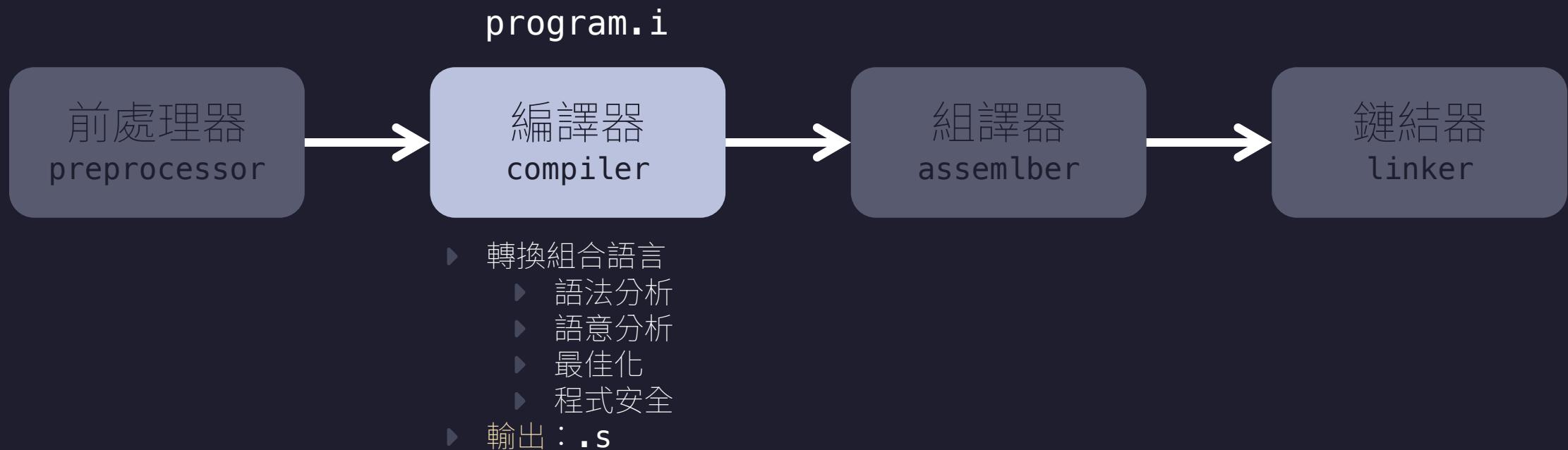
\$編譯程式

program.c

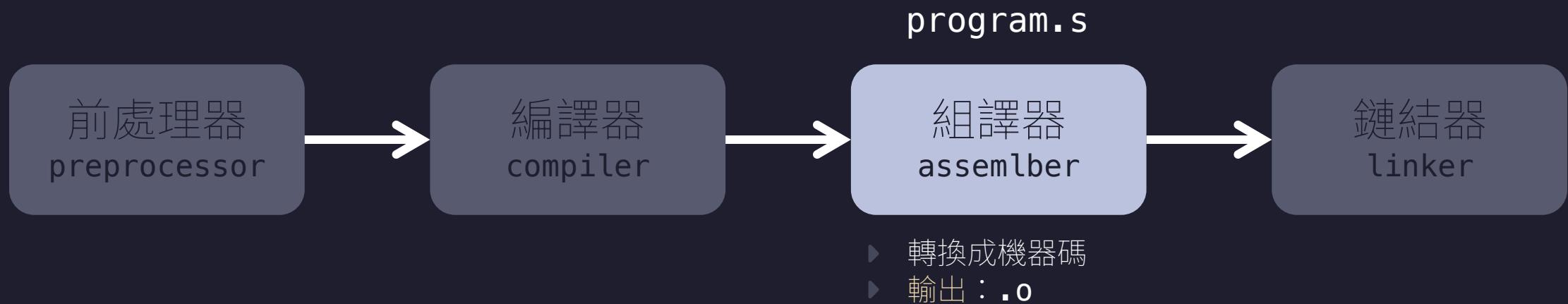


- ▶ 引入標頭檔
- ▶ 巨集替換
- ▶ 條件編譯
- ▶ 移除註解
- ▶ 輸出：`.c`

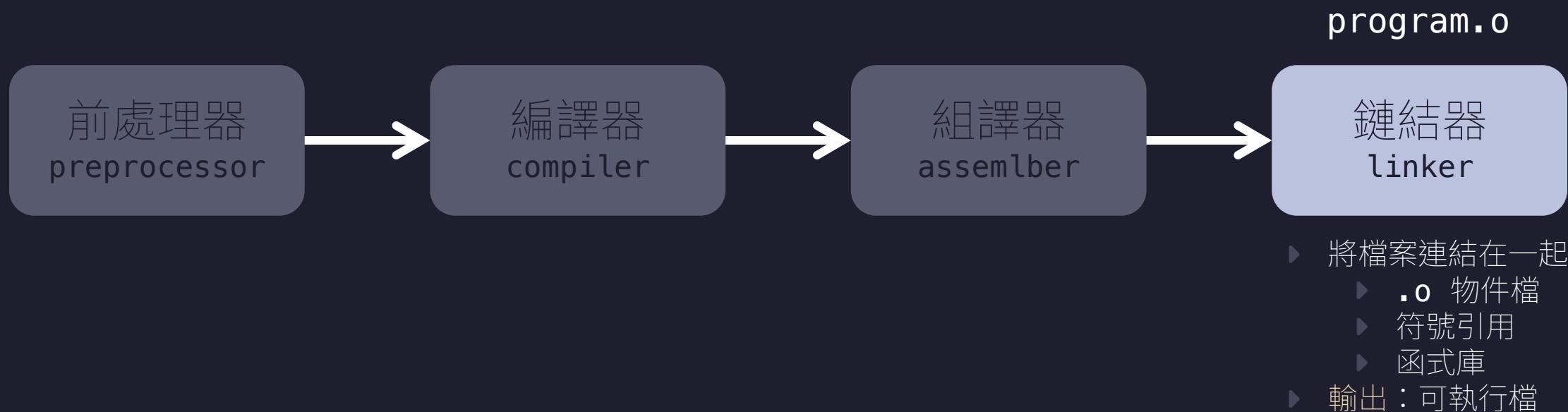
\$編譯程式



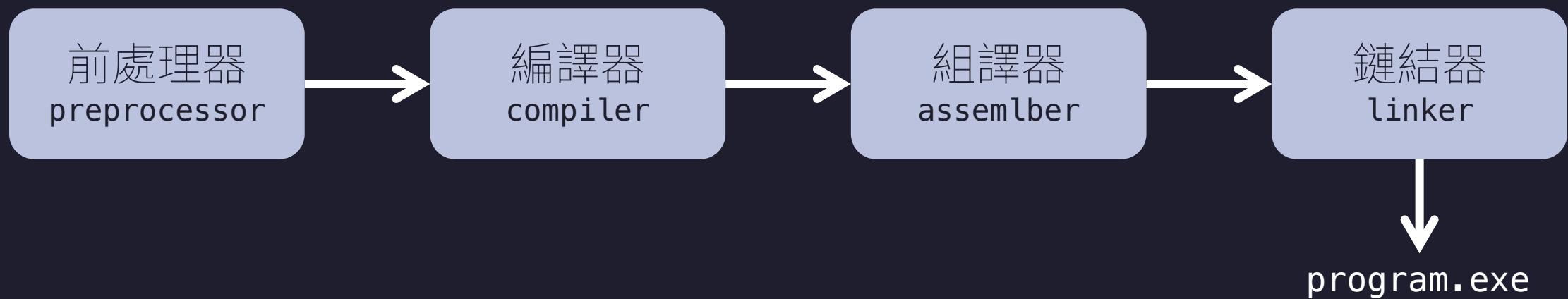
\$編譯程式



\$編譯程式



\$編譯程式



\$gcc

- ▶ GCC(GNU Compiler Collection)
 - ▶ gcc : C 編譯器
 - ▶ g++ : C++ 編譯器
 - ▶ gfortran : Fortran 編譯器
 - ▶ gobjc : Objective-C 編譯器
- ▶ 自由軟體

\$gcc

- ▶ GCC(GNU Compiler Collection)
 - ▶ gcc : C 編譯器
 - ▶ g++ : C++ 編譯器
 - ▶ gfortran : Fortran 編譯器
 - ▶ gobjc : Objective-C 編譯器
- ▶ 自由軟體

\$Lab 1-1 暖身：編譯C程式

- ▶ 把 `hello.c` 編譯成可執行檔並執行

- ▶ `gcc program.c -o program` # 從 `program.c` 編譯成 `program`
- ▶ `gcc program.c` # 從 `program.c` 編譯成 `a.out/a.exe`
- ▶ `./program`

\$C 程式的錯誤類型

- ▶ Syntax Errors 語法錯誤
- ▶ Runtime Errors 執行期錯誤
- ▶ Logical Errors 邏輯錯誤
- ▶ Linked Errors 連結錯誤
- ▶ Semantic Errors 語意錯誤

\$Syntax Error

- ▶ 不符合該程式語言的語法
- ▶ Compiler 會告訴你哪裡有問題
- ▶ 參閱 [C99 規格書](#)
- ▶ 沒加分號、括號沒刮

```
#include <stdio.h>
int main() {
    int x = 10
    printf("%d\n", x);
    return 0;
}
```

\$Semantic Errors

- ▶ 語法正確
- ▶ 但是語意有問題
- ▶ 賦值給一個表達式

```
#include <stdio.h>

int main() {
    int a = 5, b = 10;
    (a + b) = 15;
    return 0;
}
```

\$Linked Errors

- ▶ 連結過程中的錯誤
- ▶ 無法將目標檔連結成執行檔
- ▶ 使用未定義函式

```
#include <stdio.h>
int main() {
    print("hello.\n");
    return 0;
}
```

\$Runtime Errors

- ▶ Compiler 不會報錯
- ▶ 程式異常終止
- ▶ 除以零、陣列索引超出範圍

```
#include <stdio.h>

int main() {
    int a = 10, b = 0;
    int c = a / b;
    printf("%d\n", c);
    return 0;
}
```

\$Logical Errors

- ▶ 使用者程式寫錯

```
#include <stdio.h>

int main() {
    int a = 5, b = 10;
    int average = a + b / 2;
    printf("Average: %d\n", average);
    return 0;
}
```

\$Lab 1-2：編譯C程式

- ▶ 編譯 `string.c` 並執行

- ▶ `gcc program.c -o program` # 從 `program.c` 編譯成 `program`
- ▶ `gcc program.c` # 從 `program.c` 編譯成 `a.out/a.exe`
- ▶ `./program`

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

int main(int argc, char *argv[]) {
    char str[] = "Hello, world!\n";
    printf("%s\n", &str);
    return 0;
}
```

\$Lab 1-2：編譯C程式

- ▶ 編譯 `string.c` 並執行

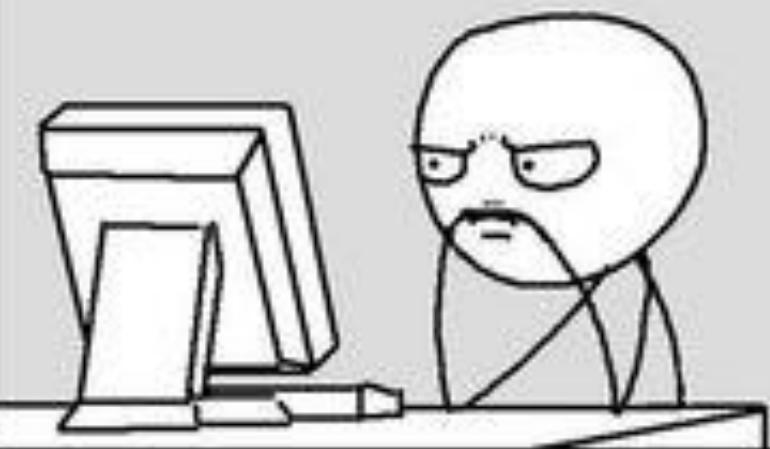
```
$ gcc string.c -o string
string.c: In function 'main'
string.c:5:14: warning: format ??s??expects argument of type 'char *' but argument 2
has type 'char (*)[15]' [-Wformat=]
 5 |     printf("%s\n", &str);
    |     ^~~~~~
    |     | char (*)[15]
    |     char *
```

\$Lab 1-2：編譯C程式

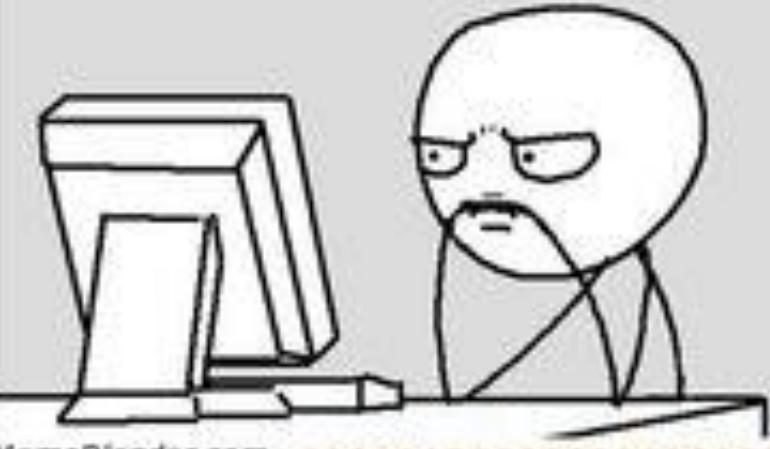
- ▶ 編譯 `string.c` 並執行

```
$ ./string  
Hello, world!
```

It doesn't work..... why?



It works..... why?



\$gcc

- ▶ Compiler 很強
- ▶ 善用 -Wall 選項

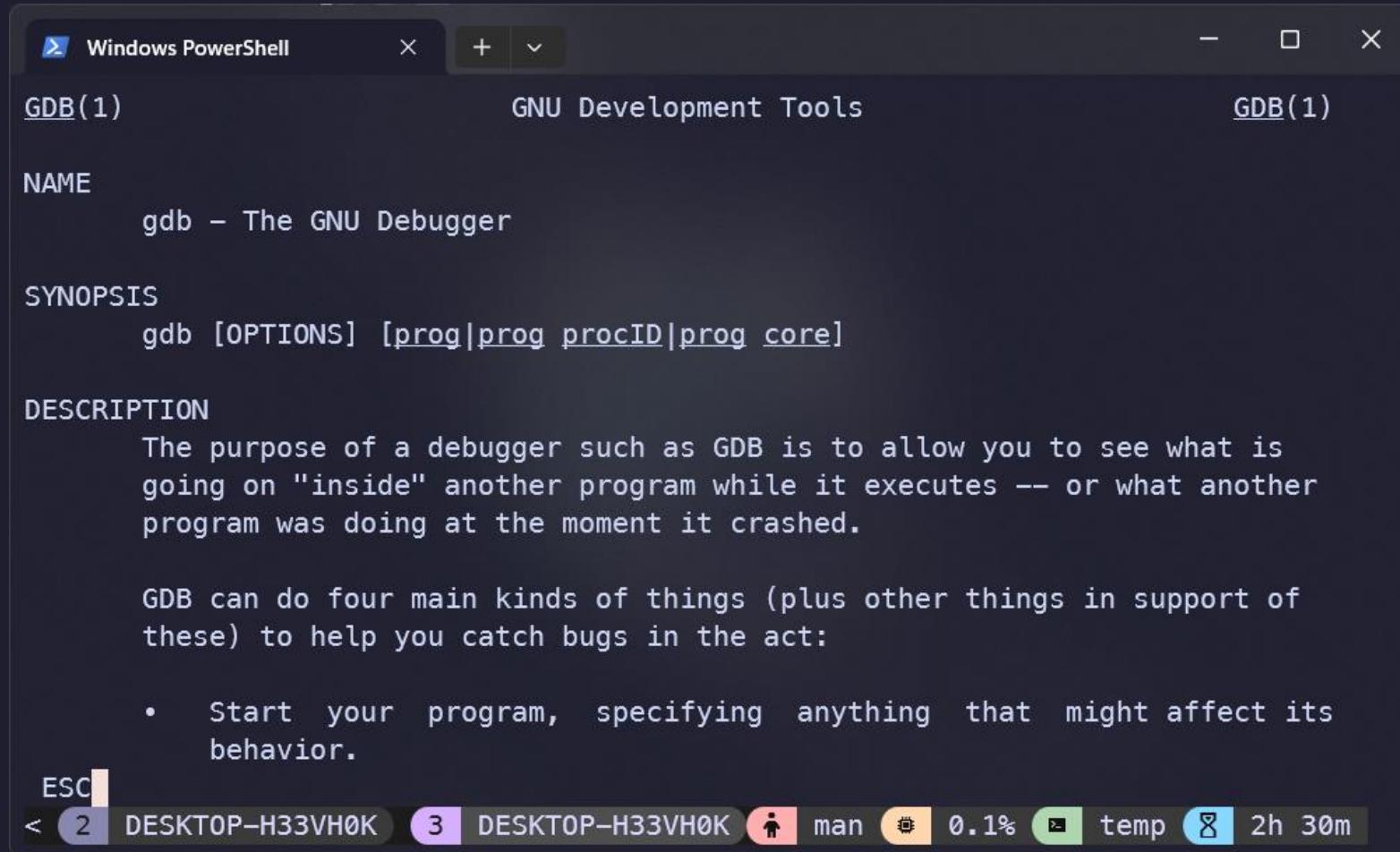


\$GDB要解決的問題

- ▶ gcc 編譯完不會報錯
- ▶ 我明明照著演算法寫啊怎麼會錯
- ▶ 為什麼遞迴跑一跑 SIGSEGV

02 使用**GDB**除錯

\$GDB



A screenshot of a Windows PowerShell window titled "Windows PowerShell". The title bar also includes "GNU Development Tools" and "GDB(1)". The main content of the window is the man page for GDB, which includes sections for NAME, SYNOPSIS, and DESCRIPTION, along with a bulleted list of features. The bottom of the window shows a taskbar with several icons and status information.

```
Windows PowerShell
GDB(1)          GNU Development Tools          GDB(1)

NAME
gdb - The GNU Debugger

SYNOPSIS
gdb [OPTIONS] [prog|prog procID|prog core]

DESCRIPTION
The purpose of a debugger such as GDB is to allow you to see what is
going on "inside" another program while it executes -- or what another
program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of
these) to help you catch bugs in the act:

• Start your program, specifying anything that might affect its
behavior.

ESC
< 2 DESKTOP-H33VH0K 3 DESKTOP-H33VH0K man 0.1% temp 2h 30m
```

\$GDB

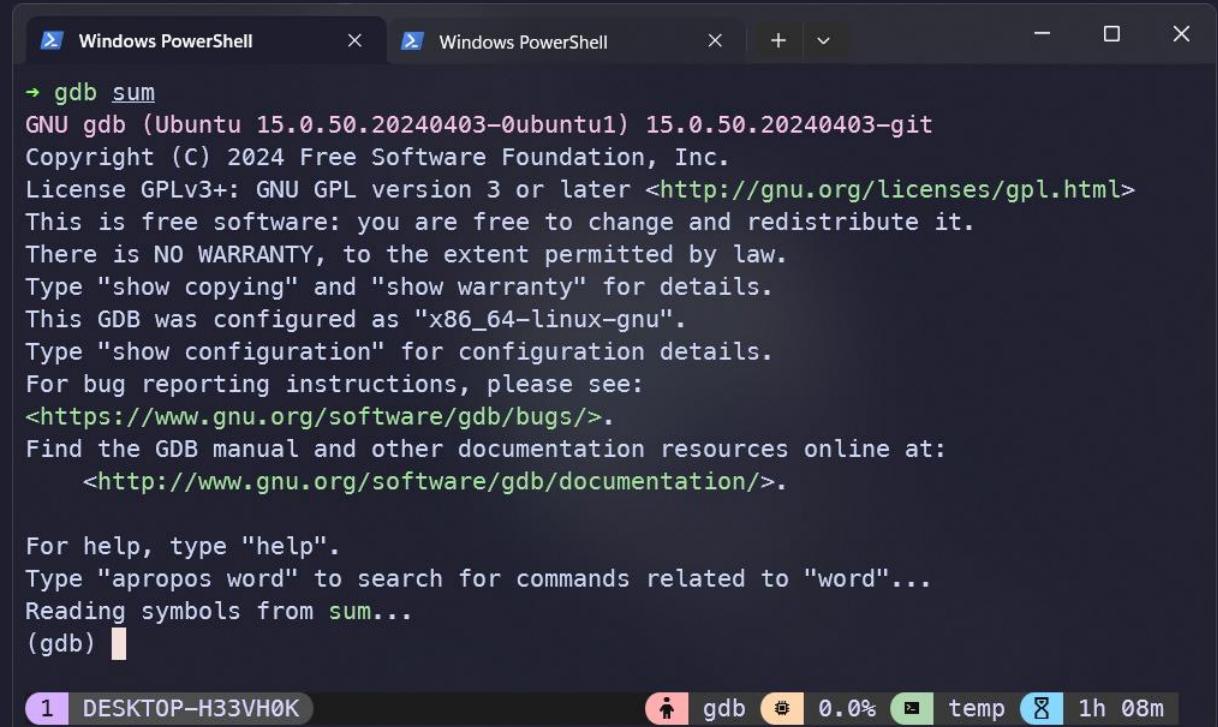
- ▶ `gdb <program>`
- ▶ `gdb <program> <pid>`
- ▶ `gdb -p <pid>`

\$Lab 2-1 : gdb 基本操作

- ▶ 編譯 sum.c 並使用 gdb 開啟

依序嘗試以下指令：

- ▶ run
- ▶ disas main
- ▶ layout split
- ▶ start
- ▶ next
- ▶ next
- ▶ step
- ▶ continue

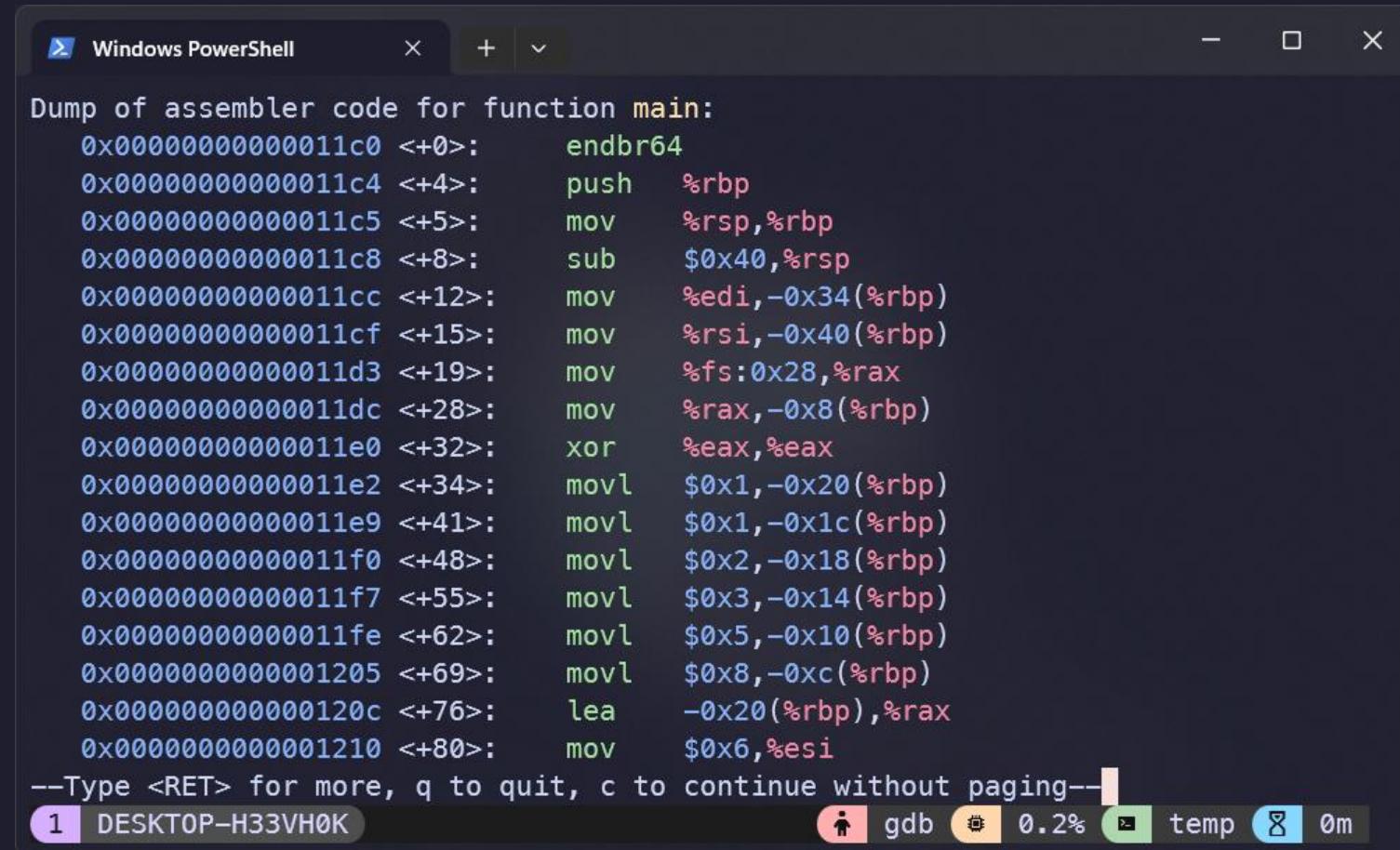


The screenshot shows a terminal window with two tabs, both titled "Windows PowerShell". The left tab contains the command "gdb sum" followed by the GDB startup message. The right tab is empty. At the bottom of the terminal window, there is a taskbar with several icons and a status bar showing "1 DESKTOP-H33VH0K", "gdb", "0.0%", "temp", "1h 08m", and other system information.

```
→ gdb sum
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from sum...
(gdb) █
```

\$Lab 2-1 : gdb 基本操作



Dump of assembler code for function main:

```
0x00000000000011c0 <+0>:    endbr64
0x00000000000011c4 <+4>:    push   %rbp
0x00000000000011c5 <+5>:    mov    %rsp,%rbp
0x00000000000011c8 <+8>:    sub    $0x40,%rsp
0x00000000000011cc <+12>:   mov    %edi,-0x34(%rbp)
0x00000000000011cf <+15>:   mov    %rsi,-0x40(%rbp)
0x00000000000011d3 <+19>:   mov    %fs:0x28,%rax
0x00000000000011dc <+28>:   mov    %rax,-0x8(%rbp)
0x00000000000011e0 <+32>:   xor    %eax,%eax
0x00000000000011e2 <+34>:   movl   $0x1,-0x20(%rbp)
0x00000000000011e9 <+41>:   movl   $0x1,-0x1c(%rbp)
0x00000000000011f0 <+48>:   movl   $0x2,-0x18(%rbp)
0x00000000000011f7 <+55>:   movl   $0x3,-0x14(%rbp)
0x00000000000011fe <+62>:   movl   $0x5,-0x10(%rbp)
0x0000000000001205 <+69>:   movl   $0x8,-0xc(%rbp)
0x000000000000120c <+76>:   lea    -0x20(%rbp),%rax
0x0000000000001210 <+80>:   mov    $0x6,%esi
```

--Type <RET> for more, q to quit, c to continue without paging--

1 DESKTOP-H33VH0K gdb 0.2% temp 0m

\$Lab 2-1 : gdb 基本操作



\$Symbol

- ▶ 變數名稱、函式名稱、行號...
- ▶ `gdb -g`

\$基本指令

- ▶ **help/h** 檢查指令的用法
- ▶ **quit/q** 退出 gdb
- ▶ **run/r** 執行程式
- ▶ **start** 執行並停在 main 函式
- ▶ **next/n** 執行下一行程式碼（不進入函數）
- ▶ **step/s** 執行下一行程式碼（進入函數）
- ▶ **finish/fin** 執行直到 return
- ▶ **continue/c** 繼續執行程式，直到下一個中斷點
- ▶ **print/p <variable>** 印出變數的值

\$圖形化介面

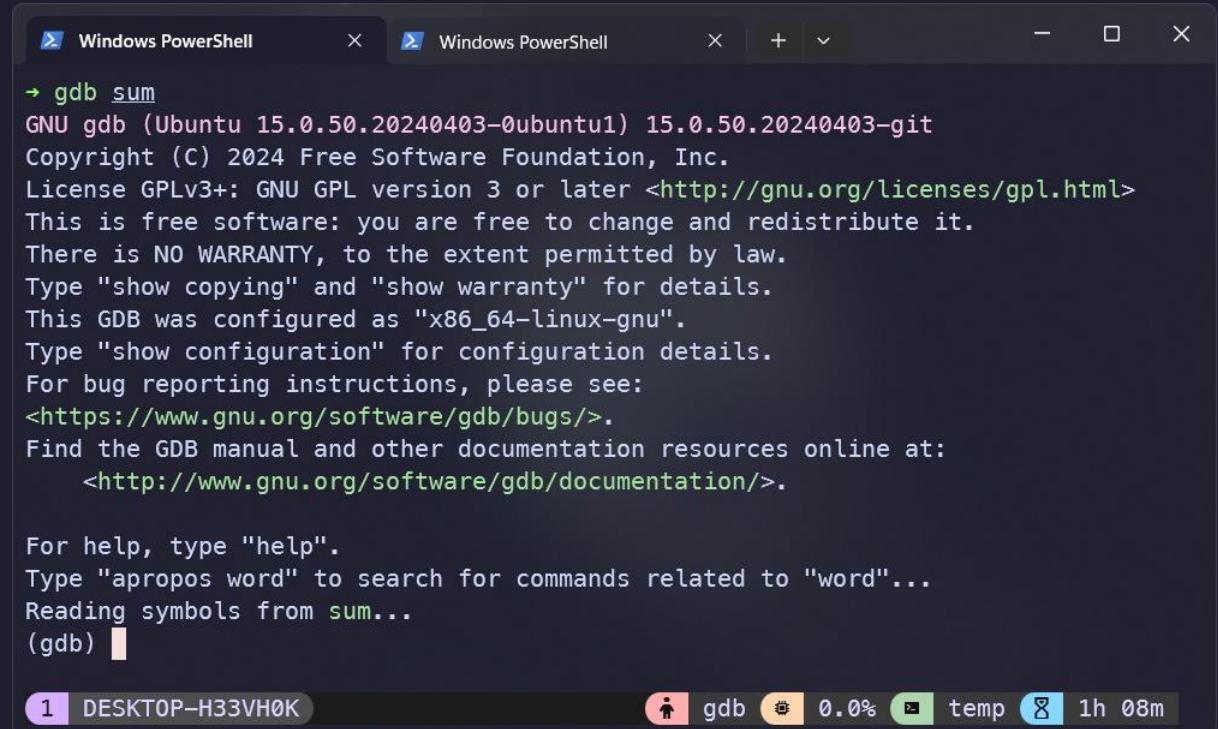
- ▶ **gdb -tui**
- ▶ **layout <option>**
 - ▶ **src** 原始碼
 - ▶ **asm** 組合語言
 - ▶ **regs** 原始碼/組合語言+暫存器
 - ▶ **split** 原始碼+組合語言
- ▶ 快捷鍵
 - ▶ **Ctrl+L** 重新整理
 - ▶ **Ctrl+A** 退出 TUI 模式

\$Lab 2-1 : gdb 基本操作

- ▶ 重新用 `-g` 選項編譯 `sum.c` 並使用 `gdb` 開啟

依序嘗試以下指令：

- ▶ `layout split` 看原始碼和組合語言
- ▶ `start` 執行程式
- ▶ `s` 進入 `arr_sum`
- ▶ `p` 觀察 `sum` 值的變化
- ▶ 使用 `fin` 跳出 `arr_sum` 或 `c` 繼續執行



```
→ gdb sum
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from sum...
(gdb) █
```

1 DESKTOP-H33VH0K gdb 0.0% temp 1h 08m

03 程式流程與斷點

\$ 點斷黑

Break point

- ▶ 時間暫停



\$ 斷點

Break point

- ▶ break/b <locspec>
- ▶ b <行號>
- ▶ b <函數名稱>
- ▶ b *<記憶體位址>
- ▶ break <中斷位置> if <條件>
- ▶ info breakpoints/break
- ▶ delete <斷點編號>
- ▶ enable <斷點編號>
- ▶ disable <斷點編號>

參考資料



Break point

- ▶ 示範

\$Lab 2-1 : gdb 基本操作

- ▶ 重新用 `-g` 選項編譯 `sum.c` 並使用 `gdb` 開啟

依序嘗試以下指令：

- ▶ `layout split` 看原始碼和組合語言
- ▶ `start` 執行程式
- ▶ `s` 進入 `arr_sum`
- ▶ `p` 觀察 `sum` 值的變化
- ▶ 使用 `fin` 跳出 `arr_sum` 或 `c` 繼續執行

```
gdb sum
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from sum...
(gdb)
```

\$Lab 3-1：建立斷點

- ▶ 編譯 `rand.c` 作答
- ▶ 這隻程式會使用 `rand()` 生成一個數字儲存到變數中，請使用 `gdb` 的 `print` 指令，找出生成的數字是多少。
- ▶ Hint：要記得下斷點，不然不會停

\$Lab 3-1：建立幽點

- ▶ 編譯 `rand.c` 作答
- ▶ `set` 指令可以改變執行中程式的變數的值，使用方法如下。嘗試使用這個指令，讓程式最後輸出的 `number` 是 `114514`。
- ▶ `set <var>=<value>`
- ▶ Hint : `set sum=0xdeadbeef`

\$流程控制

- ▶ **next/n(i)** 執行下一行程式碼（不進入函數）
- ▶ **step/s(i)** 執行下一行程式碼（進入函數）
- ▶ **finish/fin** 執行直到 `return`
- ▶ **continue/c** 繼續執行程式，直到下一個中斷點
- ▶ **until/u <locspec>** 執行直到指定的位置

\$Lab 3-1：流程控制

- ▶ 編譯 **fib.c** 作答
- ▶ 實作兩種類型的函式計算 **Fibonacci**，其中計算出了一點小問題。請嘗試使用 **gdb** 除錯，修正邏輯錯誤的地方。

04 gdbinit 與插件

\$gdbinit

- ▶ GDB 的設定腳本
- ▶ 啟動時會自動讀取並執行
- ▶ 可以使用 Python
- ▶ 網路上有很多人寫好的

```
# 設定 GDB 的提示符號  
set prompt \033[31mgdb \$ \33[0  
  
# 開啟歷史紀錄儲存功能  
set history save on  
  
# 設定顯示陣列的最大長度  
set print elements 200  
  
# 設定顯示結構體的方式  
set print object on  
set print pretty on  
  
# 自動載入符號檔  
set auto-solib-add on
```

\$GDB-Dashboard

The screenshot shows a GDB session running in a Windows PowerShell window. The session is set up to break at the start of the gcd function in gcd.c:13. TheRegisters pane displays the current register values, showing that the program has hit a breakpoint. TheSource pane shows the C code for the gcd function, with line 13 highlighted. TheVariables pane shows the arguments and local variables: argc = 1, argv = 0x7fffffff898: 47 '/', loc a = 32767, b = -10088, result = 32767. The bottom status bar indicates the session is named 'gdb' and has been running for 0.1% of a second, with 2 hours and 33 minutes remaining.

```
[1] break at 0x0000555555551c7 in gcd.c:13 for main hit 1 time
[2] break once at 0x0000555555551c7 in gcd.c:13 for -qualified main hit 1 time
Expressions
History
Memory
Registers
    rax 0x0000555555551b4    rbx 0x00007fffffff898    rcx 0x000055555557dc0    rdx 0x00007fffffff8a8    rsi 0x00007fffffff898
    rdi 0x0000000000000001    rbp 0x00007fffffff770    rsp 0x00007fffffff750    r8 0x0000000000000000    r9 0x00007ffff7fcfa380
    r10 0x00007fffffff490    r11 0x0000000000000203    r12 0x0000000000000001    r13 0x0000000000000000    r14 0x000055555557dc0
    r15 0x00007ffff7ffd000    rip 0x0000555555551c7    eflags [ PF IF ]    cs 0x00000033    ss 0x0000002b
    ds 0x00000000    es 0x00000000    fs 0x00000000    gs 0x00000000    fs_base 0x00007ffff7d9f740
    gs_base 0x0000000000000000
Source
8     return a ? gcd(b % a, a) : b;
9 }
10
11 int main(int argc, char *argv[])
12 {
13     int a = 25;
14     int b = 60;
15     int result = gcd(a, b);
16     printf("[+] gcd(%d, %d) = %d\n", a, b, result);
17     return 0;
Stack
[0] from 0x0000555555551c7 in main+19 at gcd.c:13
Threads
[1] id 129712 name gcd from 0x0000555555551c7 in main+19 at gcd.c:13
Variables
arg argc = 1, argv = 0x7fffffff898: 47 '/'
loc a = 32767, b = -10088, result = 32767
>>> 1 DESKTOP-H33VH0K
gdb 0.1% temp 2h 33m
```

\$GDB-Dashboard

- ▶ 視覺化顯示資訊
- ▶ 很適合新手入門

The screenshot shows a Windows PowerShell window running a GDB session. The session is attached to a process named 'gcd' at address 0x0000555555551c7. The GDB interface includes tabs for Expressions, History, Memory, Registers, Source, Stack, Threads, and Variables. The Registers tab displays CPU register values, and the Source tab shows the assembly code for the gcd function. The Variables tab shows the arguments argc=1, argv=0x7fffffd898, and the local variable result=32767.

```
[1] break at 0x0000555555551c7 in gcd.c:13 for main hit 1 time
[2] break once at 0x0000555555551c7 in gcd.c:13 for -qualified main hit 1 time
--- Expressions ---
--- History ---
--- Memory ---
--- Registers ---
rax 0x0000555555551b4    rbx 0x00007fffffd898    rcx 0x0000555555557c0    rdx 0x00007fffffd8a8    rsi 0x00007fffffd898
rdi 0x0000000000000001    rbp 0x00007fffffd770    rsp 0x00007fffffd750    r8 0x0000000000000000    r9 0x00007fffffca380
r10 0x00007fffffd490    r11 0x0000000000000203    r12 0x0000000000000001    r13 0x0000000000000000    r14 0x000055555557dc0
r15 0x00007fffffd000    rip 0x0000555555551c7    eflags [ PF IF ]    cs 0x00000033    ss 0x0000002b
ds 0x00000000    es 0x00000000    fs 0x00000000    gs 0x00000000    gs_base 0x00007ffff7d9f740
gs_base 0x0000000000000000
--- Source ---
8      return a ? gcd(b % a, a) : b;
9 }
10
11 int main(int argc, char *argv[])
12 {
13     int a = 25;
14     int b = 60;
15     int result = gcd(a, b);
16     printf("[%i] gcd(%d, %d) = %d\n", a, b, result);
17     return 0;
--- Stack ---
[0] from 0x0000555555551c7 in main+19 at gcd.c:13
--- Threads ---
[1] id 129712 name gcd from 0x0000555555551c7 in main+19 at gcd.c:13
--- Variables ---
arg argc = 1, argv = 0x7fffffd898 47 '/'
loc a = 32767, b = -10088, result = 32767
>>> 1 DESKTOP-H33VH0K
gdb 0.1% temp 2h 33m
```

\$PEDA

```
Windows PowerShell      + | - | X
R9 : 0x7ffff7fca380 (<_dl_fini>:          endbr64)
R10: 0x7fffffff490 --> 0x800000
R11: 0x203
R12: 0x1
R13: 0x0
R14: 0x555555555dc0 --> 0x555555555100 (<__do_global_dtors_aux>:      endbr64)
R15: 0x7ffff7ffd000 --> 0x7ffff7ffe2e0 --> 0x555555554000 --> 0x10102464c457f
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x5555555551bc <main+8>:    sub    rsp,0x20
0x5555555551c0 <main+12>:   mov    DWORD PTR [rbp-0x14],edi
0x5555555551c3 <main+15>:   mov    QWORD PTR [rbp-0x20],rsi
=> 0x5555555551c7 <main+19>:  mov    DWORD PTR [rbp-0xc],0x19
0x5555555551ce <main+26>:   mov    DWORD PTR [rbp-0x8],0x3c
0x5555555551d5 <main+33>:   mov    edx, DWORD PTR [rbp-0x8]
0x5555555551d8 <main+36>:   mov    eax, DWORD PTR [rbp-0xc]
0x5555555551db <main+39>:   mov    esi,edx
[-----stack-----]
0000| 0x7fffffff750 --> 0x7fffffff898 --> 0x7fffffffdbc2 ("/mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab3-2/gcd")
0008| 0x7fffffff758 --> 0x1f7fe5af0
0016| 0x7fffffff760 --> 0x7fffffff850 --> 0x55555555060 (<_start>:  endbr64)
0024| 0x7fffffff768 --> 0x7fffffff898 --> 0x7fffffffdbc2 ("/mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab3-2/gcd")
0032| 0x7fffffff770 --> 0x7fffffff810 --> 0x7fffffff870 --> 0x0
0040| 0x7fffffff778 --> 0x7ffff7dcc1ca (<_libc_start_call_main+122>:  mov    edi,eax)
0048| 0x7fffffff780 --> 0x7fffffff7c0 --> 0x555555557dc0 --> 0x555555555100 (<__do_global_dtors_aux>: endbr64)
0056| 0x7fffffff788 --> 0x7fffffff898 --> 0x7fffffffdbc2 ("/mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab3-2/gcd")
[-----]
Legend: code, data, rodata, value

Breakpoint 1, main (argc=0x1, argv=0x7fffffff898) at gcd.c:13
13      int a = 25;
gdb-peda$ 1 DESKTOP-H33VH0K
  1  gcd  0.1%  temp  1h 39m
```

\$PEDA

- ▶ Python Exploit Development Assistance for GDB
- ▶ 強化 GDB 原生指令，更直覺的 暫存器顯示、反組譯輸出、記憶體檢視
- ▶ 介面較為簡潔、功能基本
- ▶ 適合CTF新手挑戰 pwn

```
Windows PowerShell x + v
R9 : 0x7ffff7fca380 (<_dlsym_r>: endbr64)
R10: 0x7fffffff490 --> 0x800000
R11: 0x203
R12: 0x1
R13: 0x0
R14: 0x555555555100 --> 0x555555555100 (<_do_global_dtors_aux>: endbr64)
R15: 0x7ffff7ffd000 --> 0x7ffff7ffe2e0 --> 0x555555554000 --> 0x10102464c457f
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x5555555551bc <main+8>:    sub   rsp,0x20
0x5555555551c0 <main+12>:   mov    DWORD PTR [rbp-0x14],edi
0x5555555551c3 <main+15>:   mov    DWORD PTR [rbp-0x20],rsi
=> 0x5555555551c7 <main+19>:  mov    DWORD PTR [rbp-0xc],0x19
0x5555555551ce <main+26>:  mov    DWORD PTR [rbp-0x8],0x3c
0x5555555551d5 <main+33>:  mov    edx,DWORD PTR [rbp-0x8]
0x5555555551d8 <main+36>:  mov    eax,DWORD PTR [rbp-0xc]
0x5555555551db <main+39>:  mov    esi,edx
[-----stack-----]
0000| 0x7ffff7fd750 --> 0x7ffff7fd898 --> 0x7ffff7fdabc2 ('/mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab3-2/gcd')
0008| 0x7ffff7fd758 --> 0x17fffe5af0
0016| 0x7ffff7fd760 --> 0x7ffff7fd850 --> 0x555555555000 (<_start>: endbr64)
0024| 0x7ffff7fd768 --> 0x7ffff7fd898 --> 0x7ffff7fdabc2 ('/mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab3-2/gcd')
0032| 0x7ffff7fd770 --> 0x7ffff7fd810 --> 0x7ffff7fd870 --> 0x0
0040| 0x7ffff7fd778 --> 0x7ffff7dc1ca (<_libc_start_call_main+122>: mov edi,eax)
0048| 0x7ffff7fd780 --> 0x7ffff7fd7e0 --> 0x555555555100 (<_do_global_dtors_aux>: endbr64)
0056| 0x7ffff7fd788 --> 0x7ffff7fd898 --> 0x7ffff7fdabc2 ('/mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab3-2/gcd')
[-----]
Legend: code, data, rodata, value

Breakpoint 1, main (argc=0x1, argv=0x7ffff7fd898) at gcd.c:13
13      int a = 25;
gdb-peda$ 1 DESKTOP-H33VHOK
1 gcd 0 0.1% temp 1h 39m
```

\$ GEF

The screenshot shows the GEF debugger interface running in a Windows PowerShell window. The assembly view at the top displays several memory addresses and their corresponding assembly instructions, with some lines highlighted in green. The source code view below shows the C code for a gcd function, with the current instruction at line 13 highlighted in green. The bottom part of the interface contains command-line controls for GDB, including a status bar showing 'gdb' and '0.1%'.

```
Windows PowerShell x + v - o x
0x00007fffffd778 |+0x0028: 0x00007ffff7dcc1ca → <__libc_start_main+007a> mov edi, eax
0x00007fffffd780 |+0x0030: 0x00007fffffd7c0 → 0x0000555555557dc0 → 0x000055555555100 → <__do_global_dtors_aux+0000> endbr64
0x00007fffffd788 |+0x0038: 0x00007fffffd898 → 0x00007fffffdcbc2 → "/mnt/c/Users/Yuto/Desktop/gdbTutorial/labs/lab3-2[...]" code:x86:64
0x55555555551bc <main+0008>    sub   rsp, 0x20
0x55555555551c0 <main+000c>    mov    DWORD PTR [rbp-0x14], edi
0x55555555551c3 <main+000f>    mov    QWORD PTR [rbp-0x20], rsi
●→ 0x55555555551c7 <main+0013>  mov    DWORD PTR [rbp-0xc], 0x19
0x55555555551ce <main+001a>    mov    DWORD PTR [rbp-0x8], 0x3c
0x55555555551d5 <main+0021>    mov    edx, DWORD PTR [rbp-0x8]
0x55555555551d8 <main+0024>    mov    eax, DWORD PTR [rbp-0xc]
0x55555555551db <main+0027>    mov    esi, edx
0x55555555551dd <main+0029>    mov    edi, eax
source:gcd.c+13
8     return a ? gcd(b % a, a) : b;
9 }
10
11 int main(int argc, char *argv[])
12 {
13     // a=0xffff
→ 13     int a = 25;
14     int b = 60;
15     int result = gcd(a, b);
16     printf("[+] gcd(%d, %d) = %d\n", a, b, result);
17     return 0;
18 }
threads
[#0] Id 1, Name: "gcd", stopped 0x555555551c7 in main (), reason: BREAKPOINT
trace
[#0] 0x555555551c7 → main(argc=0x1, argv=0x7fffffd898)
gef> 1 DESKTOP-H33VH0K
gdb 0.1% temp 1h 38m
```

\$gef

- ▶ GDB Enhanced Features
- ▶ Python 3 支援
- ▶ 多種 CPU 架構支援(x86, x86_64, ARM, MIPS, PowerPC 等)

The screenshot shows the GEF debugger running in a Windows PowerShell window. The assembly view at the top displays the main function's entry point at address 0x00007fffffd778, followed by several instructions involving memory operations and register moves. A red arrow points to the instruction at address 0x555555551c7, which corresponds to the source code line 13: `int a = 25;`. The source code view below shows the C code for the gcd function, with line 13 highlighted. The bottom status bar shows the current session is 'gef>' and the host computer is 'DESKTOP-H33VH0K'.

```
Windows PowerShell
0x00007fffffd778 +0x0028: 0x00007ffffd7dc1ca → <__libc_start_main+007a> mov edi, eax
0x00007fffffd780 +0x0030: 0x00007fffffd7c0 → 0x0000555555557dc0 → 0x000055555555100 → <_do_global_dtors_aux+0000> endbr64
0x00007fffffd788 +0x0038: 0x00007fffffd898 → 0x00007fffffd898 → "/mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab3-2[...]"
code:x86:64
source:gcd.c+13
8     return a ? gcd(b % a, a) : b;
9 }
10
11 int main(int argc, char *argv[])
12 {
13     // a=0xffff
14     int a = 25;
15     int b = 60;
16     int result = gcd(a, b);
17     printf("[%+] gcd(%d, %d) = %d\n", a, b, result);
18 }
```

[#0] Id 1, Name: "gcd", stopped 0x555555551c7 in main (), reason: BREAKPOINT
[#0] 0x555555551c7 → main(argc=0x1, argv=0x7fffffd898)

gef> 1 DESKTOP-H33VH0K

\$Pwndbg

The screenshot shows the Pwndbg debugger interface running in a Windows PowerShell window. The interface is divided into several sections:

- SOURCE (CODE):** Shows the C code for a gcd function. Line 13 is highlighted with a red arrow, indicating the current instruction being executed.
- STACK:** Displays a memory dump of the stack. The stack grows from bottom to top. The current instruction at address 0x555555551ea is shown as mov edx, dword ptr [rbp - 8]. The stack dump shows frames from 00:0000 to 07:0038, each containing the current instruction and its operands.
- BACKTRACE:** Shows the call stack, listing the addresses and names of the functions called during the program's execution.
- Prompt:** The command line prompt is "pwndbg>" followed by a small terminal icon.
- Status Bar:** Shows the current state: "1 DESKTOP-H33VH0K", "gdb", "0.2%", "temp", and "1h 38m".

\$Pwndbg

- ▶ 專為 **CTF** 與漏洞開發設計，提供自動化利用工具
- ▶ 偵測 **ASLR**、**NX**、**Canary**、**PIE** 等安全機制
- ▶ 內建 **Heap** 分析工具
- ▶ 適合**CTF**競賽選手

The screenshot shows the Pwndbg debugger running in a Windows PowerShell window. The assembly code pane displays the main function of a C program, which calculates the greatest common divisor (GCD) of two integers, `a` and `b`. The code includes a check for `a` being zero and prints the result to the console. The stack dump pane shows the current state of the stack, including memory addresses and values. The backtrace pane shows the call stack, starting from the main function and leading up to the `__libc_start_main` entry point. The bottom status bar indicates the debugger is running in a temporary session.

```
Windows PowerShell
In file: /mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab3-2/gcd.c:13
8     return a ? gcd(b % a, a) : b;
9 }
10
11 int main(int argc, char *argv[])
12 {
13     int a = 25;
14     int b = 0;
15     int result = gcd(a, b);
16     printf("[%] gcd(%d, %d) = %d\n", a, b, result);
17     return 0;
18 }

[ STACK ]
00:0000| rsp 0x7fffffff750 -> 0x7fffffff898 -> 0x7fffffffdbc2 -> '/mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab3-2/gcd'
01:0008|-018 0x7fffffff750 -> 0x1fffe5af0
02:0010|-010 0x7fffffff760 -> 0x7fffffff850 -> 0x555555555060 (_start) -> endbr64
03:0018|-008 0x7fffffff768 -> 0x7fffffff898 -> 0x7fffffffdbc2 -> '/mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab3-2/gcd'
04:0020| rbp 0x7fffffff770 -> 0x7fffffff810 -> 0x7fffffff870 -> 0
05:0028|+008 0x7fffffff778 -> 0x7ffff7dcc1ca (_libc_start_call_main+122) -> mov edi, eax
06:0030|+010 0x7fffffff780 -> 0x7fffffff7c0 -> 0x555555557dc0 (_do_global_dtors_aux) -> 0x55555555100 (_do_global_dtors_aux) -> endbr64
07:0038|+018 0x7fffffff788 -> 0x7fffffff898 -> 0x7fffffffdbc2 -> '/mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab3-2/gcd'

[ BACKTRACE ]
0 0x5555555551c7 main+19
1 0x7ffff7dcc1ca __libc_start_call_main+122
2 0x7ffffdcc28b __libc_start_main+139
3 0x555555555085 _start+3

pwndbg> 1 DESKTOP-H33VH0K
gdb 0.2% temp 1h 38m
```

\$Lab 4-1：安裝 gdb-dashboard

- ▶ 下載 gdb-dashboard 的 [.gdbinit](#) 到 ~/.gdbinit
- ▶ sudo apt install python3-pygment
- ▶ pip install pygments

\$gdb-dashboard

- ▶ 專為 **CTF** 與漏洞開發設計，提供自動化利用工具
- ▶ 偵測 **ASLR**、**NX**、**Canary**、**PIE** 等安全機制
- ▶ 內建 **Heap** 分析工具
- ▶ 適合 CTF 競賽選手

\$Lab 4-1：快樂除錯

- ▶ 編譯 gcd.c 作答
- ▶ 這個程式也出了一點小問題，請嘗試修復錯誤

05 變數檢視

\$變數檢視

- ▶ 每次都要手打 `print` 超累
- ▶ `display <expr>`
- ▶ `info display`
- ▶ `undisplay/delete display <nms>...`
- ▶ `enable display <nms>...`
- ▶ `disable display <nms>...`

\$變數檢視

\$變數檢視

06 程式競賽的應用

\$測資

- ▶ 用 Linux 的 pipeline 機制
 - ▶ `./program < test.in > output.txt`
 - ▶ `diff test.out output.txt`
 - ▶ `vimdiff test.out output.txt`
 - ▶ `nvim -d test.out output.txt`

\$測資

The screenshot shows a terminal window titled "Windows PowerShell" with a dark theme. On the left, the code for echo.c is displayed:

```
C echo.c x |   ≡ in.txt x
7 #include <stdio.h>
6
5 int main(int argc, char *argv[]) {
4 |     char input[4096] = {0};
3 |     while (scanf(format: "%s", input) != EOF)
2 |         printf(format: "%s\n", input);
1 |     return 0;
8 }
```

On the right, the terminal output shows the build process and a diff command:

```
→ gcc echo.c -o echo
→ ./echo < in.txt > out.txt
→ diff in.txt out.txt
3c3,4
< hello ttussc
---
> hello
> ttussc

○ /tmp
→
```

The status bar at the bottom provides additional information:

NORMAL ➤ C echo.c > f main ^wl ⌂ 17 Bot 8:1 ⌂ 16:16
1 DESKTOP-H33VH0K 2 DESKTOP-H33VH0K

zsh 0.1% temp 2h 18m

\$測資

```
Windows PowerShell
in.txt      x |   out.txt      x
3 hello
2 ttussc
1 hello ttussc
4 linux
4 hello
3 ttussc
2 hello
1 ttussc
5 linux
```

NORMAL ➤ in.txt 1 DESKTOP-H33VH0K 2 DESKTOP-H33VH0K nvim ^[⊞ 17 Bot 4:5 ⊞ 16:19 0.1% temp 2h 21m

\$Lab 6-1：餵測資

- ▶ 編譯 `echo.c`
- ▶ 測資：`test.in`、測資輸出：`test.out`
- ▶ 修改程式使其輸出吻合測資

\$測資

- ▶ 剩下的就等你去實戰了

07 函式呼叫

08 記憶體傾印

Happing Hacking

wayne71112@gmail.com