

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

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 update
- ▶ sudo apt upgrade
- ▶ sudo apt install git gcc vim

\$ 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
- ▶ cd gdb_tutorial
- ▶ 更新：
 - ▶ git fetch

SIGSEGV時在做什麼？ 有沒有**GDB**？可以來除錯嗎？

講師：Yuto

\$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

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;
}
```

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 Search results Ccc Build log Build messages CppCheck CppCheck mes

```
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!
```

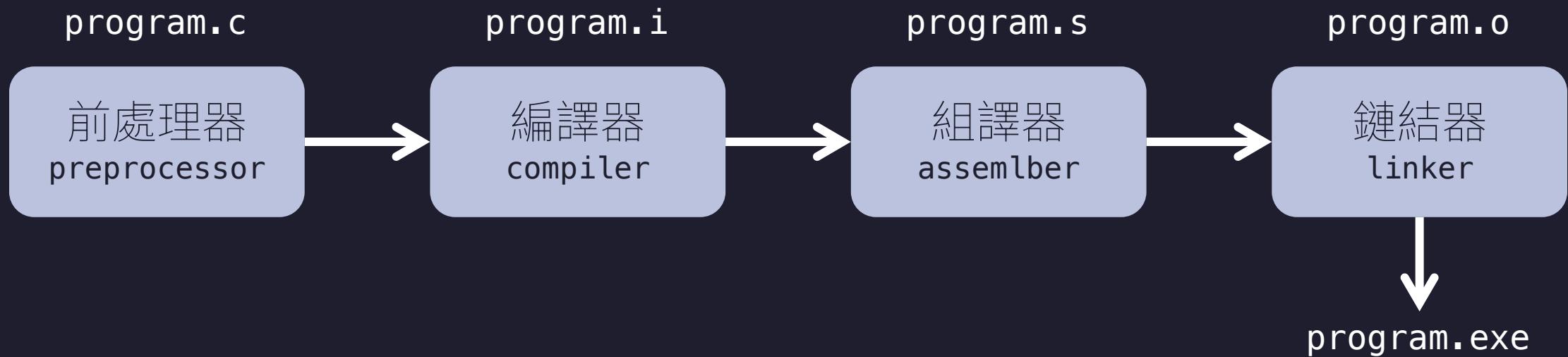
C:\SoftpediaTest\Softpedia C++ test\Softpi Windows (CR+LF) WINDOWS-1252 Line 1, Column 20 Insert Read/Write default

Celinglass



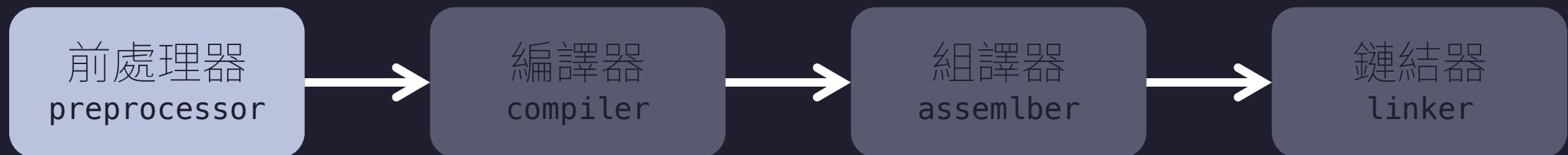
老師我的程式碼怎麼編譯那麼久？

\$編譯程式



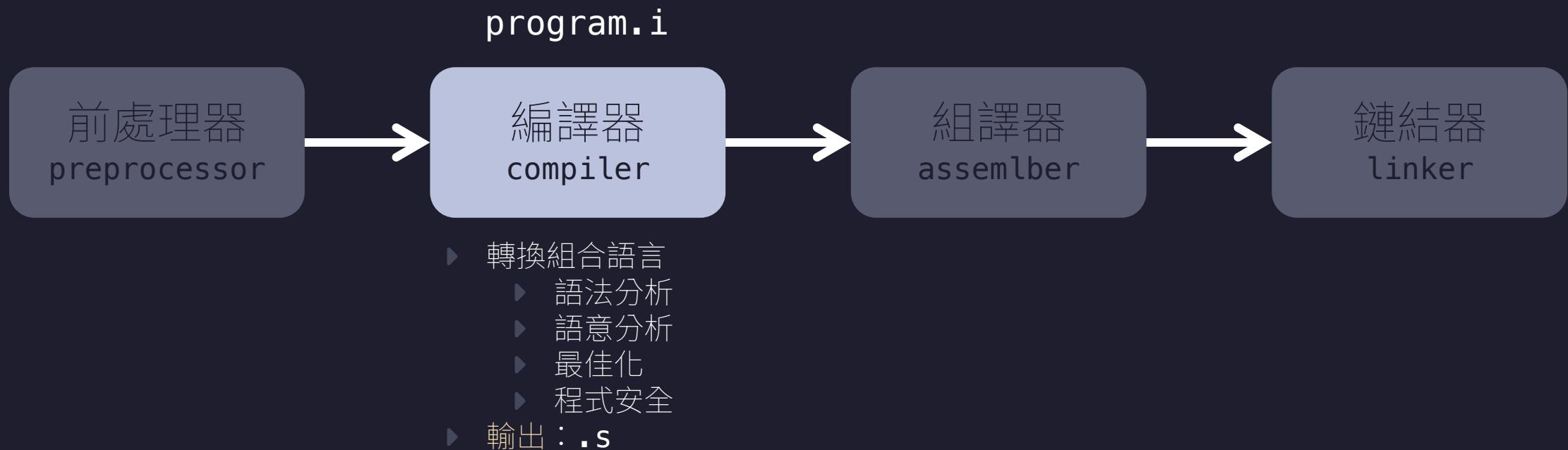
\$編譯程式

program.c

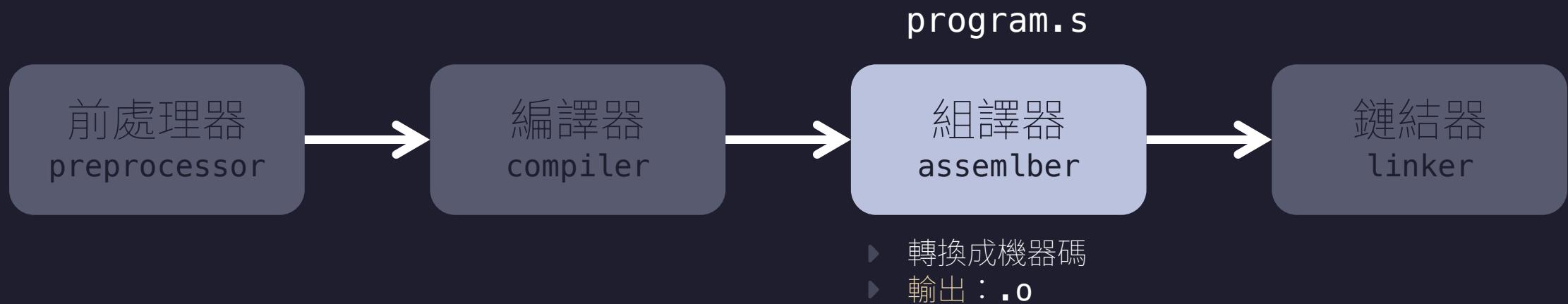


- ▶ 引入標頭檔
- ▶ 巨集替換
- ▶ 條件編譯
- ▶ 移除註解
- ▶ 輸出：`.c`

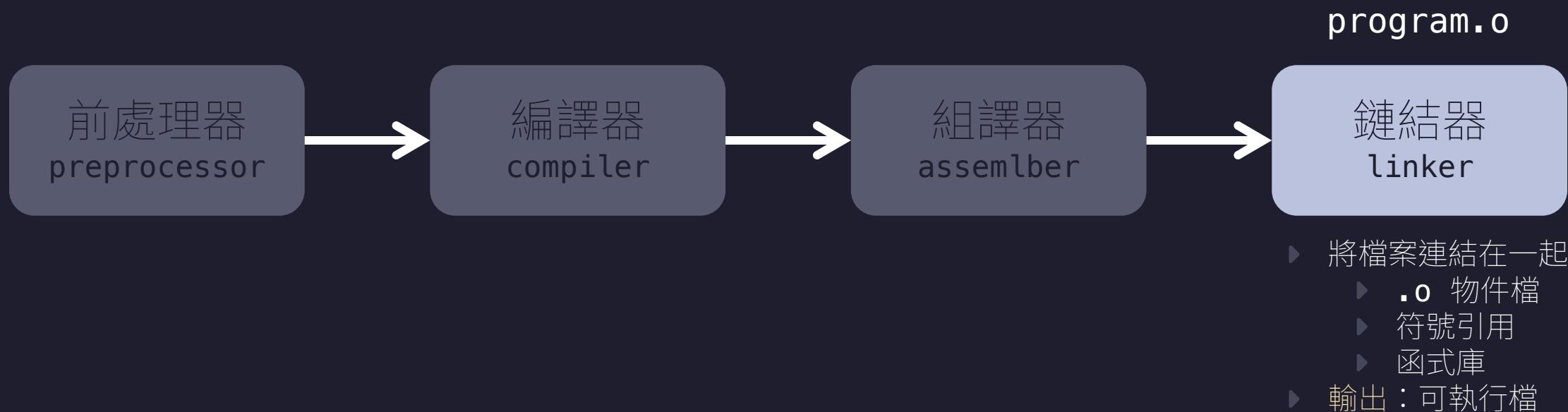
\$編譯程式



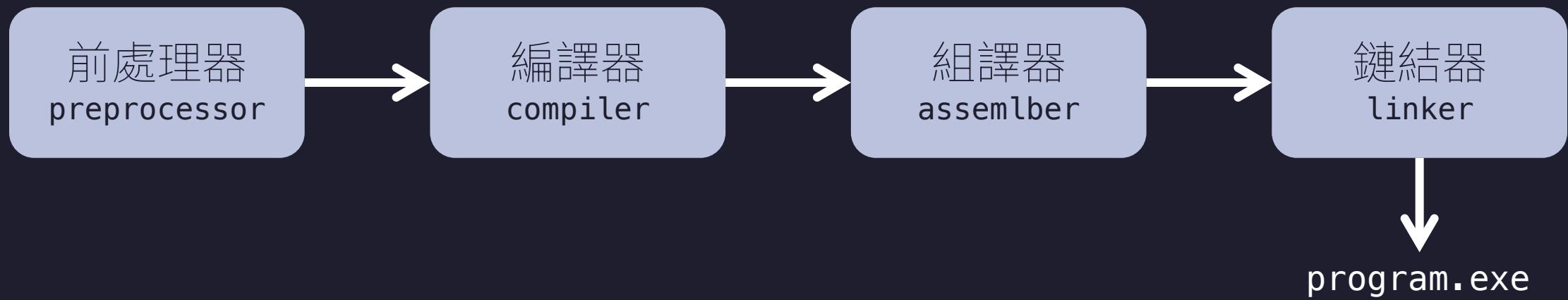
\$編譯程式



\$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

\$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

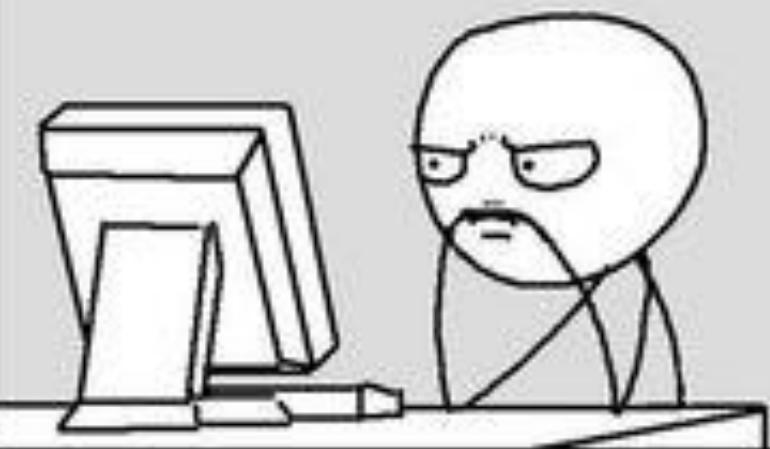
```
$ gcc string.c -o string
string.c: In function 'main'
string.c:5:14: warning: format ??s??expects argument of type 'char *' but argument 2
has type 'char (*)[15]' [-Wformat=]
 5 |     printf("%s\n", &str);
    |     ^~~~~
    |     | char (*)[15]
    |     char *
```

\$Lab 1-2：編譯C程式

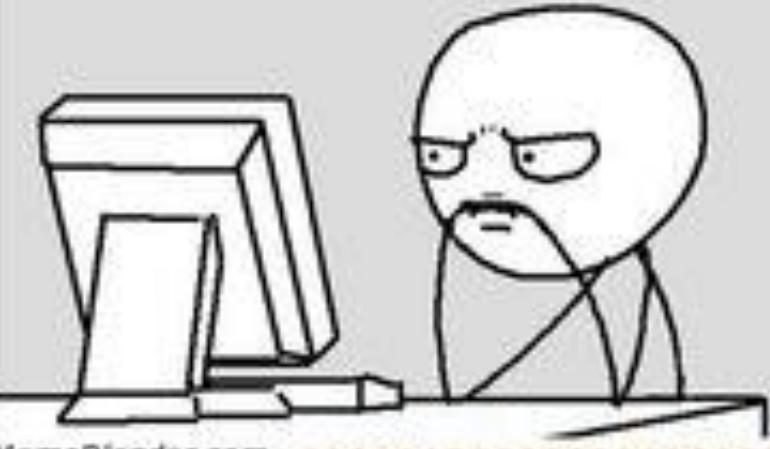
- ▶ 編譯 `string.c` 並執行

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

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



It works..... why?



\$gcc

- ▶ Compiler 很強
- ▶ 善用 -Wall 選項

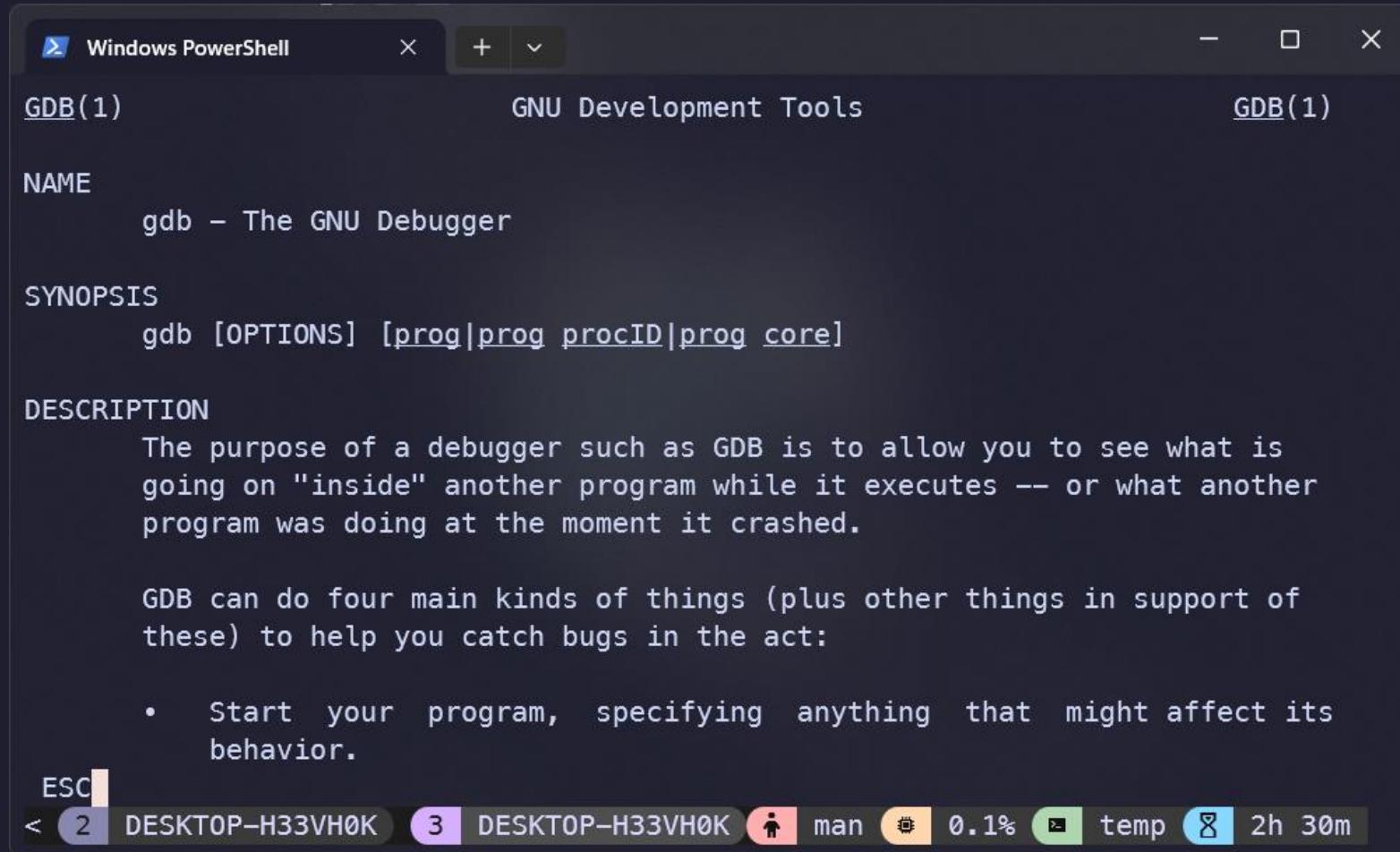


\$GDB要解決的問題

- ▶ 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" with a dark theme. The title bar also displays "GNU Development Tools" and "GDB(1)". The main content of the window is the man page for GDB:

```
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.
```

The bottom of the window shows a taskbar with several icons and status information: "ESC", "2 DESKTOP-H33VH0K", "3 DESKTOP-H33VH0K", a user icon, "man", a progress bar at "0.1%", a "temp" folder icon, and "2h 30m".

\$GDB

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

\$基本指令

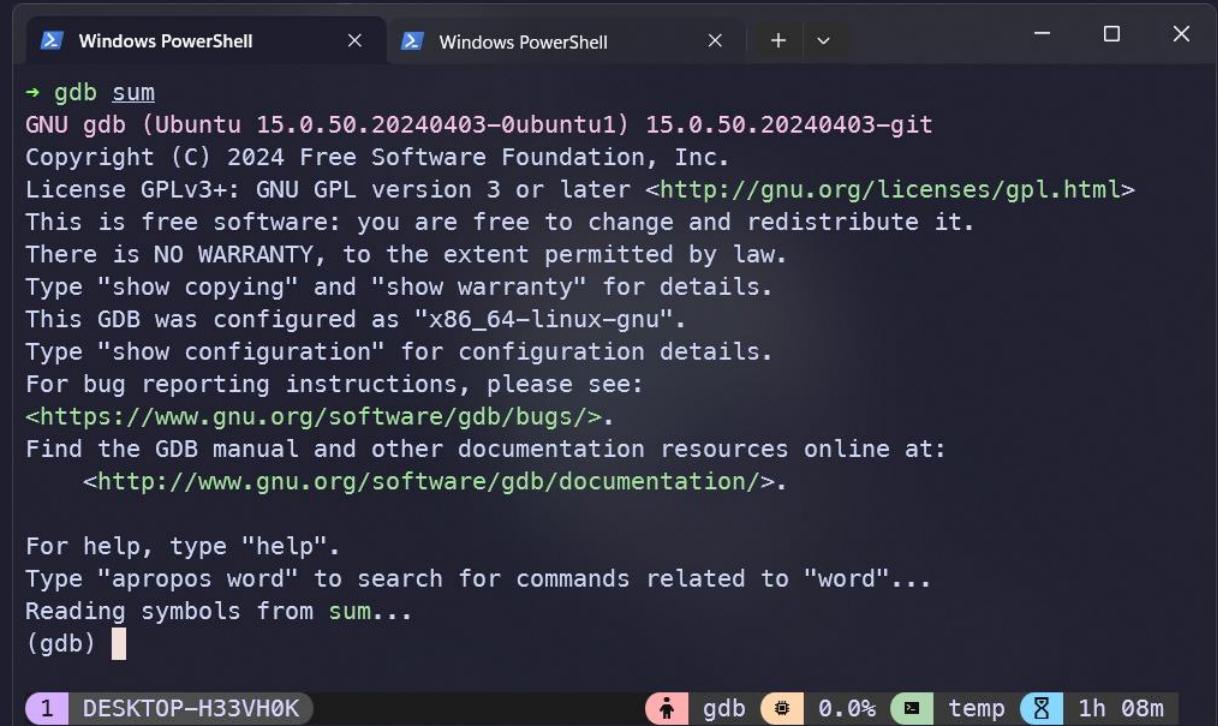
- ▶ **help/h** 檢查指令的用法
- ▶ **quit/q** 退出 gdb
- ▶ **run/r** 執行程式

\$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：

- ▶ help disas
- ▶ disas main
- ▶ run
- ▶ quit

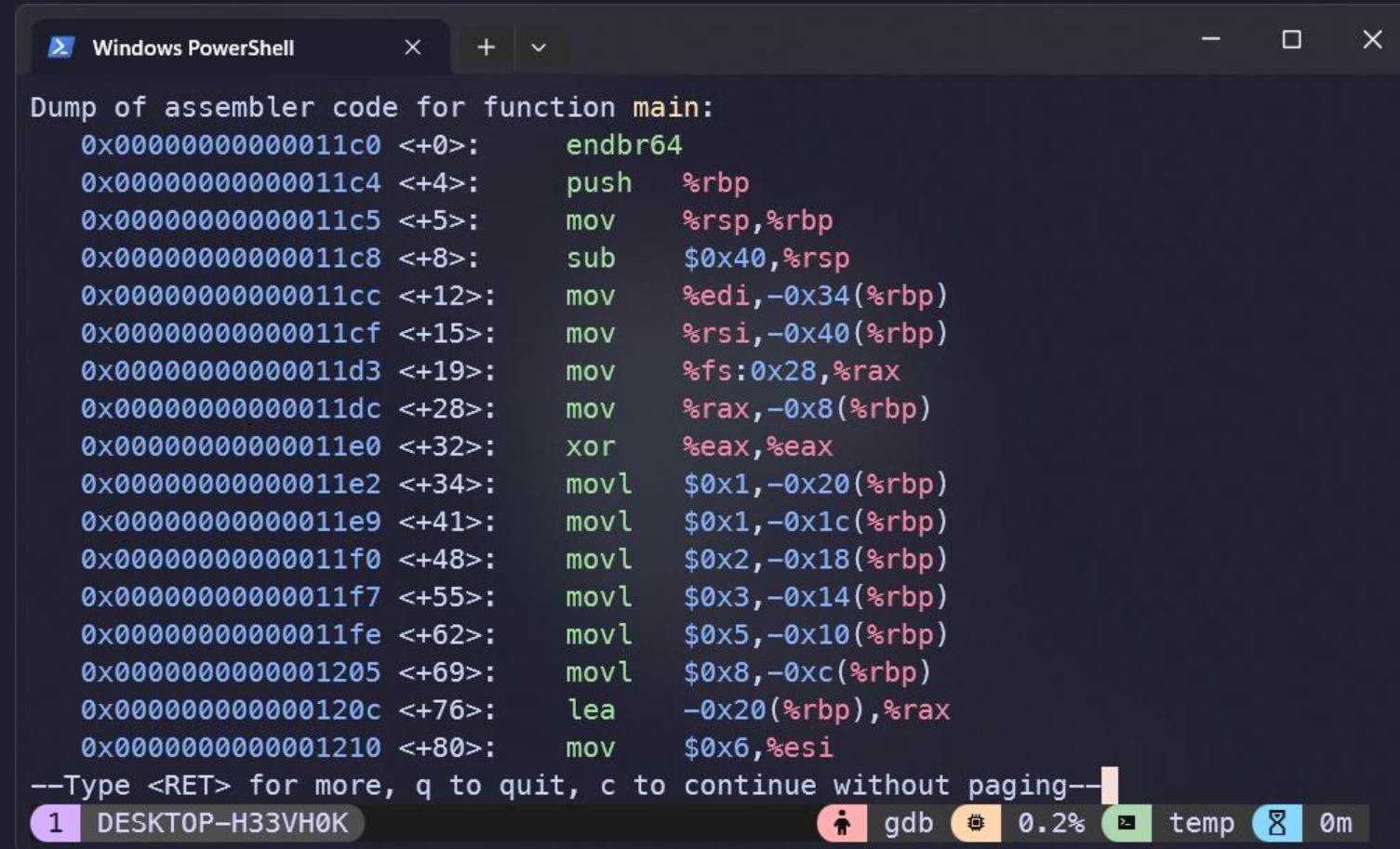


```
→ gdb sum
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the 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

\$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...
- ▶ gcc -g

\$原始碼

- ▶ `list` : 顯示原始碼 / 接續往下顯示
- ▶ `list <func>` : 顯示函式原始碼
- ▶ `list <linenum>` : 顯示某行數的原始碼
- ▶ `list +` : 接續往下顯示
- ▶ `list -` : 往上顯示

\$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：

- ▶ `list` 瀏覽全部的原始碼
- ▶ `list arr_sum`
- ▶ `list main`
- ▶ `list +`
- ▶ `list -`

A screenshot of a Windows desktop environment. At the top, there are two taskbar icons for "Windows PowerShell". Below them, a terminal window titled "gdb sum" is open, displaying the GNU GDB version information and configuration details. The terminal window shows the following text:

```
→ 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) █
```

The taskbar also shows other open applications: "temp" and "1h 08m".

\$圖形化介面

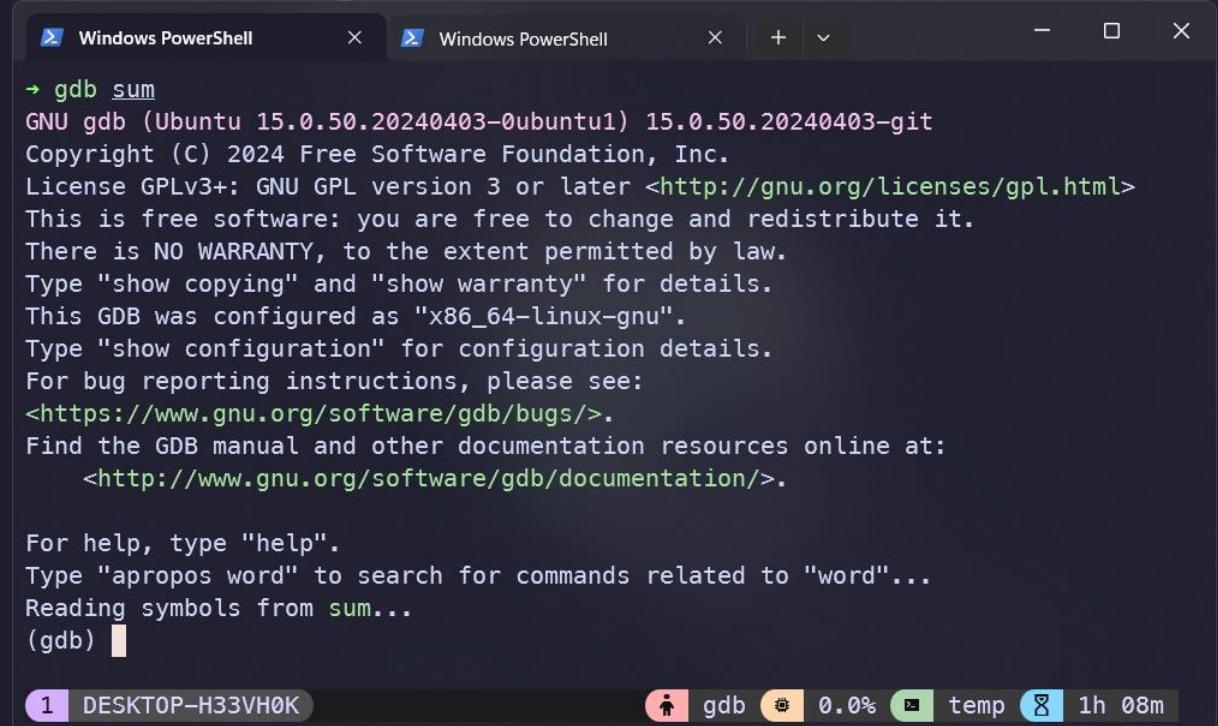
- ▶ **gdb -tui**
- ▶ **layout <option>**
 - ▶ **src** 原始碼
 - ▶ **asm** 組合語言
 - ▶ **regs** 原始碼/組合語言+暫存器
 - ▶ **split** 原始碼+組合語言
- ▶ 快捷鍵
 - ▶ **Ctrl+L** 重新整理
 - ▶ **Ctrl+X+A** 退出 TUI 模式

\$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
- ▶ 重複按 `S` 觀察執行狀況
- ▶ 使用 `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

\$流程控制

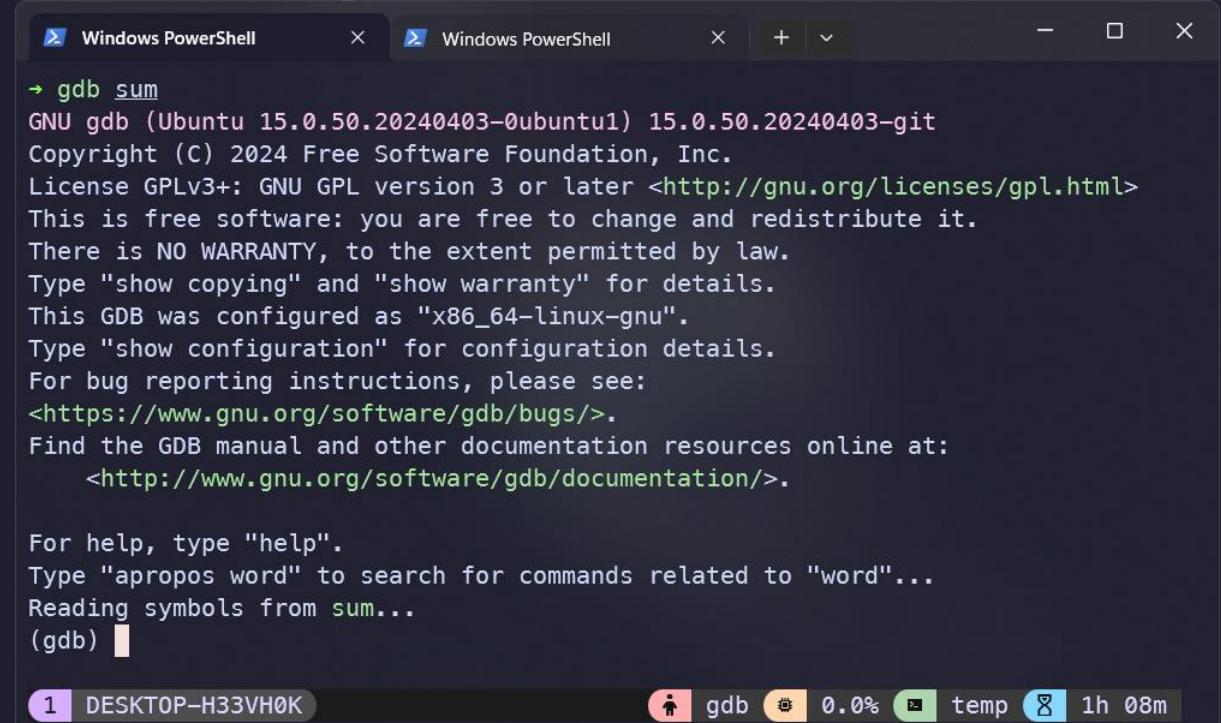
- ▶ **next/n(i)** 執行下一行程式碼（不進入函數）
- ▶ **step/s(i)** 執行下一行程式碼（進入函數）
- ▶ **finish/fin** 執行直到 `return`
- ▶ **continue/c** 繼續執行程式，直到下一個中斷點
- ▶ **until/u <locspec>** 執行直到指定的位置

- ▶ **print/p <expr>** 印出變數的值

\$Lab 3-1：硬幣投擲

▶ 用 `-g` 選項編譯 `coin.c`

- ▶ 編譯後先執行程式碼玩玩看
- ▶ 挑戰先不看程式碼，用 `gdb` 摸索程式碼的流程
 - ▶ `start`
 - ▶ `next/step`
 - ▶ `...`
- ▶ 程式說明：
 - ▶ 用統計學驗證翻硬幣的公平性
 - ▶ 調整 `myflip` 的參數調整公平性
 - ▶ 翻 10^7 次硬幣會有好看的進度條



```
→ 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

\$locspec

- ▶ Linespec
- ▶ Explicit
- ▶ Address

\$locspec

- ▶ **Linespec**
 - ▶ **linenum** : 目前檔案的第 `linenum` 行
 - ▶ **+/-offset** : 指定的位址加/減偏移量
 - ▶ **function** : 某個函式
 - ▶ **filename:** : 指定某個檔案的某個位置
- ▶ **Explicit**
- ▶ **Address**

\$locspec

- ▶ Linespec
- ▶ **Explicit**
 - ▶ 用更詳細的方式指定位置
 - ▶ **-source filename** : 指定原始檔案
 - ▶ **-function function** : 指定函式
 - ▶ **-line number** : 指定行號
- ▶ Address

\$locspec

- ▶ Linespec
- ▶ Explicit
- ▶ **Address**
 - ▶ ***address** : 這類型的前面都要加 *，跟指標不一樣
 - ▶ **expression** : C 語言位址的表達式
 - ▶ **funcaddr** : 某個 function 的位址

\$Lab 3-2：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。
-
- ▶ **1, 1, 2, 3, 5, 8, 13, 21, 34, 55...**
 - ▶ $a_n = a_{n-1} + a_{n-2}$

\$ 點斷繚黑

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

- ▶ b main

\$Lab 3-1：硬幣投擲

▶ 用 -g 選項編譯 coin.c

- ▶ 編譯後先執行程式碼玩玩看
- ▶ 挑戰先不看程式碼，用 gdb 摸索程式碼的流程
 - ▶ start
 - ▶ next/step
 - ▶ ...
- ▶ 程式說明：
 - ▶ 用統計學驗證翻硬幣的公平性
 - ▶ 調整 myflip 的參數調整公平性
 - ▶ 翻 10^7 次硬幣會有好看的進度條

```
+ 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) [REDACTED]
```

60

\$Lab 3-3：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 3-3：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`

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 'temp' and has been running for 2 hours and 33 minutes.

```
[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 set to break at address 0x0000555555551c7 in gcd.c:13. The registers pane displays various CPU registers with their addresses and values. The source pane shows the C code for the gcd function, with line 13 highlighted. The stack pane shows the call stack, and the threads pane shows a single thread. The variables pane shows the values of argc, argv, a, b, and result. The bottom status bar indicates the session ID is 1, the host is DESKTOP-H33VH0K, and the current memory usage is 0.1%.

```
[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 0x0000555555557dc0    rdx 0x00007fffffd8a8    rsi 0x00007fffffd898
rdi 0x0000000000000001    rbp 0x00007fffffd770    rsp 0x00007fffffd750    r8 0x0000000000000000    r9 0x00007fffffca380
r10 0x00007fffffd490    r11 0x00000000000000203    r12 0x0000000000000001    r13 0x0000000000000000    r14 0x0000555555557dc0
r15 0x00007fffffd000    rip 0x0000555555551c7    rflags [ PF IF ]    cs 0x00000033    ss 0x0000002b
ds 0x00000000    es 0x00000000    fs 0x00000000    gs 0x00000000    gs_base 0x00007fffffd9f740
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 = 0x7fffffd898 47 '/'
loc a = 32767, b = -10088, result = 32767
>>> 1 DESKTOP-H33VH0K
0.1% temp 2h 33m
```

\$PEDA

```
Windows PowerShell x + v - o 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_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
  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



The screenshot shows a Windows PowerShell window running a GDB session. The command `gdb-peda` is used to start the PEDA interface. The assembly code for the `main` function is displayed, showing instructions like `sub`, `mov`, and `jmp`. Registers R9 through R15 are listed at the top. The memory dump shows the stack contents, including the value `0x55555555100` at address `0x7ffff7fd898` which is identified as the breakpoint. A legend at the bottom defines the symbols: code, data, rodata, and value. The status bar at the bottom right shows the current file is `gcd` and the memory usage is 0.1%.

\$ GEF

The screenshot shows the GEF debugger 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 line numbers 8 through 18 visible. A red dot marks the current instruction at line 13. The command-line interface at the bottom shows the user's commands and the debugger's responses.

```
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 entry point and the start of the gcd function. The source view below shows the C code for the gcd function. A red dot marks the current instruction at line 13: `int a = 25;`. The status bar at the bottom shows the system is using 0.1% of memory.

```
Windows PowerShell
0x00007fffffd778 +0x0028: 0x00007ffffd7dc1ca → <__libc_start_main+007a> mov edi, eax
0x00007fffffd780 +0x0030: 0x00007fffffd7c0 → 0x0000555555555100 → <_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
0x555555551bc <main+0008> sub    rsp, 0x20
0x555555551c0 <main+000c> mov    DWORD PTR [rbp-0x14], edi
0x555555551c3 <main+000f> mov    QWORD PTR [rbp-0x20], rsi
* 0x555555551c7 <main+0013> mov    DWORD PTR [rbp-0xc1], 0x19
0x555555551ce <main+001a> mov    DWORD PTR [rbp-0x8], 0x3c
0x555555551d5 <main+0021> mov    edx, DWORD PTR [rbp-0x8]
0x555555551d8 <main+0024> mov    eax, DWORD PTR [rbp-0xc]
0x555555551db <main+0027> mov    esi, edx
0x555555551dd <main+0029> mov    edi, eax
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>
```

\$Pwndbg

The screenshot shows the Pwndbg debugger interface running in a Windows PowerShell window. The assembly code pane displays instructions from memory addresses 0x555555551ea to 0x555555551ed. The source code pane shows the C code for a gcd function, with line 13 (int a = 25;) highlighted. The stack dump pane shows the stack frame starting at 00:0000, listing registers and their values. The backtrace pane shows the call stack from address 0x555555551c7 up to _start+37. The bottom status bar indicates the session is named DESKTOP-H33VH0K, using gdb, and has been running for 0.2% of an hour (38m).

```
0x555555551ea <main+54>    mov    edx, dword ptr [rbp - 8]
0x555555551ed <main+57>    mov    eax, dword ptr [rbp - 0xc]

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 = 60;
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 -> 0x7fffffff7898 -> 0x7fffffffdbc2 ← '/mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab3-2/gcd'
01:0008|-018 0x7fffffff758 ← 0x1f7fe5af0
02:0010|-010 0x7fffffff760 -> 0x7fffffff850 -> 0x55555555060 (_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_fini_array_entry) -> 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 0x555555551c7 main+19
1 0x7ffff7dcc1ca __libc_start_call_main+122
2 0x7ffff7dcc28b __libc_start_main+139
3 0x55555555085 _start+37

pwndbg>
```

gdb 0.2% temp 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 return statement and a printf call. 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 on a desktop machine named DESKTOP-H33VH0K.

```
Windows PowerShell
0x5555555551ea <main+54>    mov    edx, dword ptr [rbp - 8]
0x5555555551ed <main+57>    mov    eax, dword ptr [rbp - 0xc]
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| rsv 0x7fffffff750 -> 0x7fffffffdb98 -> 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 -> 0x7fffffffdb98 -> 0x7fffffffdbc2 -> '/mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab3-2/gcd'
04:0020| rbp 0x7fffffff770 -> 0x7fffffff810 -> 0x7fffffffdb70 -> 0
05:0028|+008 0x7fffffff778 -> 0x7ffff7dcc1ca (_libc_start_call_main+122) -> mov edi, eax
06:0030|+010 0x7fffffff780 -> 0x7fffffff7c0 -> 0x555555557dc0 (_do_global_dtors_aux) -> 0x555555555100 (_do_global_dtors_aux) -> endbr64
07:0038|+018 0x7fffffff788 -> 0x7fffffffdb98 -> 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

- ▶ help dashboard 查看手冊
- ▶ dashboard 叫出顯示介面
- ▶ dashboard -layout 設定顯示版面
 - ▶ assembly、breakpoints、expressions
 - ▶ history、memory、registers
 - ▶ source、stack、threads、variables
 - ▶ ex:dashboard -layout breakpoints memory source stack variables



|113學年度第二學期 密碼學 Homework 1

1. 撰寫一程式計算任意兩個整數的最大公因數，請實作 Euclidean Algorithm 找最大公因數。執行結果須如上課講義 Cryptography_Lecture 2 第 25 ~ 26 頁的計算樣式。(50%)

2. 撰寫一程式計算任意兩個整數的最大公因數、以及求得 s 與 t 值。請實作 Extended Euclidean Algorithm 找最大公因、求得 s 與 t 值。執行結果須如上課講義 Cryptography_Lecture 2 第 31 ~ 33 頁的計算樣式。(50%)

1. 程式語言: C。

2. 請於程式碼中註明註解。

3. 以word撰寫操作手冊(含執行或操作過程截圖與說明)。

4. 請將 **程式碼**、**操作手冊** 壓縮，

壓縮檔命名為 **I4B59_20250305_CRYPTO_HW1**

5. 將壓縮檔上傳網路大學作業區

6. 繳交時間：2025/03/03 (三) 17:00 前 (滿分)

~ 2025/03/09 (日) 23:00 前 (補繳那題分數折半)

7. 評分重點：程式正確性。

\$密碼學

```
int gcd(int a, int b)
{
    return a ? gcd(b % a, a) : b;
}
```

\$密碼學

```
int gcd(int a, int b)
{
    printf("[*] %4d %4d %4d %4d\n", b / a, a, b, b % a);
    return a ? gcd(b % a, a) : b;
}
```

\$Lab 4-1：密碼學

- ▶ 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>...`

\$Lab 5-1：鏈結串列

- ▶ 編譯 `list.c` 作答
- ▶ 這個程式沒什麼問題，練習用 `display` 追蹤串列的值

\$Lab 5-2：氣泡排序

- ▶ 編譯 **bubble.c** 作答
 - ▶ 這個程式沒什麼問題，練習用 **display** 追蹤陣列的值
-
- ▶ Hint :
 - ▶ p *arr@len

06 程式競賽的應用

\$redirect & pipe

- ▶ < ` > : 重新導向輸入輸出
- ▶ >> : 新增在後面
 - ▶ ex : echo “hello” >> hello.txt
- ▶ | : 把左邊程式的輸出導向右邊程式的輸入
 - ▶ ex : echo “~” | cd

\$測資

- ▶ 用 Linux 的 pipe 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

```
Ubuntu
→ ./echo < test.in > out.txt
→ diff out.txt test.out
1c1,7
< helloabcdefg1234567~!@#$%^&*( )+ttussctaiwanhellottussc
\ No newline at end of file
---
> hello
> abcdefg
> 1234567
> ~!@#$%^&*( )+
> ttussc
> taiwan
> hello ttussc
\ No newline at end of file

○ /mnt/c/Users/Yuto/Desktop/gdb_tutorial/labs/lab6-1      100% | 13:34:43
→
```

1 DESKTOP-H33VH0K 2 DESKTOP-H33VH0K zsh 0.0% temp 14h 20m

\$測資

```
Windows PowerShell
```

```
in.txt      out.txt
```

```
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 0.1% temp 2h 21m
```

```
^[ ⊞ 17 < Bot 4:5 ⊞ 16:19
```

\$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

\$ 函式呼叫



\$ 函式呼叫

- ▶ **backtrace/bt**

- ▶ **frame/f**
- ▶ **up**
- ▶ **down**
- ▶ **info frame/f**
- ▶ **info args**
- ▶ **info locals**

\$Lab 7-1：很深

- ▶ 編譯 `deep.c`
- ▶ 用 `gdb` 追蹤函式呼叫過程

08 記憶體傾印

\$ 記憶體傾印

- ▶ **x/nfu addr**
 - ▶ **n:** count
 - ▶ **f:** format
 - ▶ **x:** 16 進位
 - ▶ **i:** 組合語言
 - ▶ **u:** unit
 - ▶ **b:** byte
 - ▶ **w:** word, 4 bytes

\$回饋表單



Happing Hacking

wayne71112@gmail.com