拆弹专家

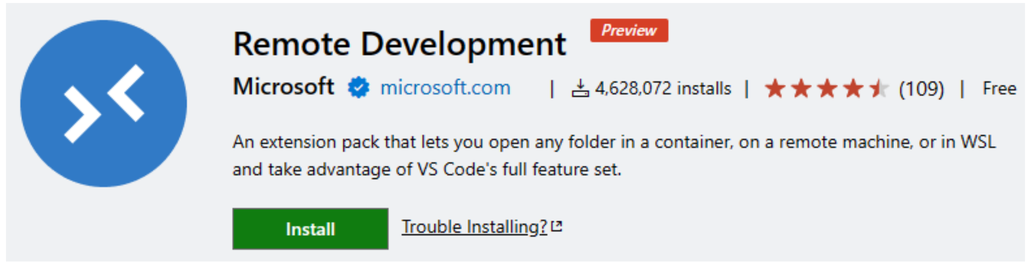
计算机系统概论实验导引(1)

张宇轩
yuxuanzh23@mails.tsinghua.edu.cn

远程实验环境使用

为了方便大家开展实验，我们提供了课程服务器资源

远程登录是后续学习、科研、工作中必备的技能
其重要性不亚于sudo apt update


Remote Development Preview
Microsoft microsoft.com | 4,628,072 installs | ★★★★☆ (109) | Free
An extension pack that lets you open any folder in a container, on a remote machine, or in WSL and take advantage of VS Code's full feature set.

本次导引会着重介绍：
Shell下的ssh登录 + 基于VS Code Remote Development开发

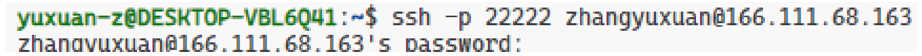
你也可以使用其它远程登录软件，比如MobaXterm等
配合命令行文本编辑器 vim 或 nano 进行开发

后续实验要求必须在课程服务器上提交，具体见每次实验的说明书
（调试类）实践题的评分也以课程服务器上的测试结果为准

SSH使用 (Shell)

在终端中输入ssh命令，输入密码登录

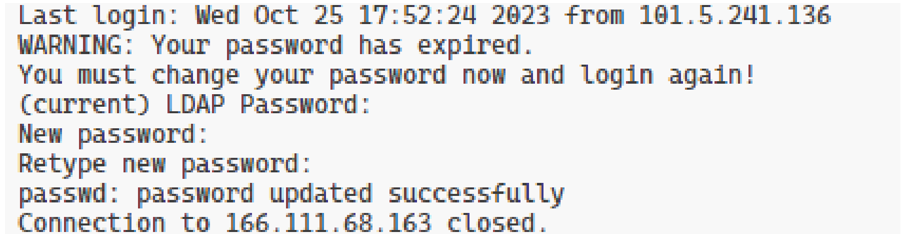
ssh -p 22222 用户名@166.111.68.163



```
yuxuan-z@DESKTOP-VBL6Q41:~$ ssh -p 22222 zhangyuxuan@166.111.68.163
zhangyuxuan@166.111.68.163's password:
```

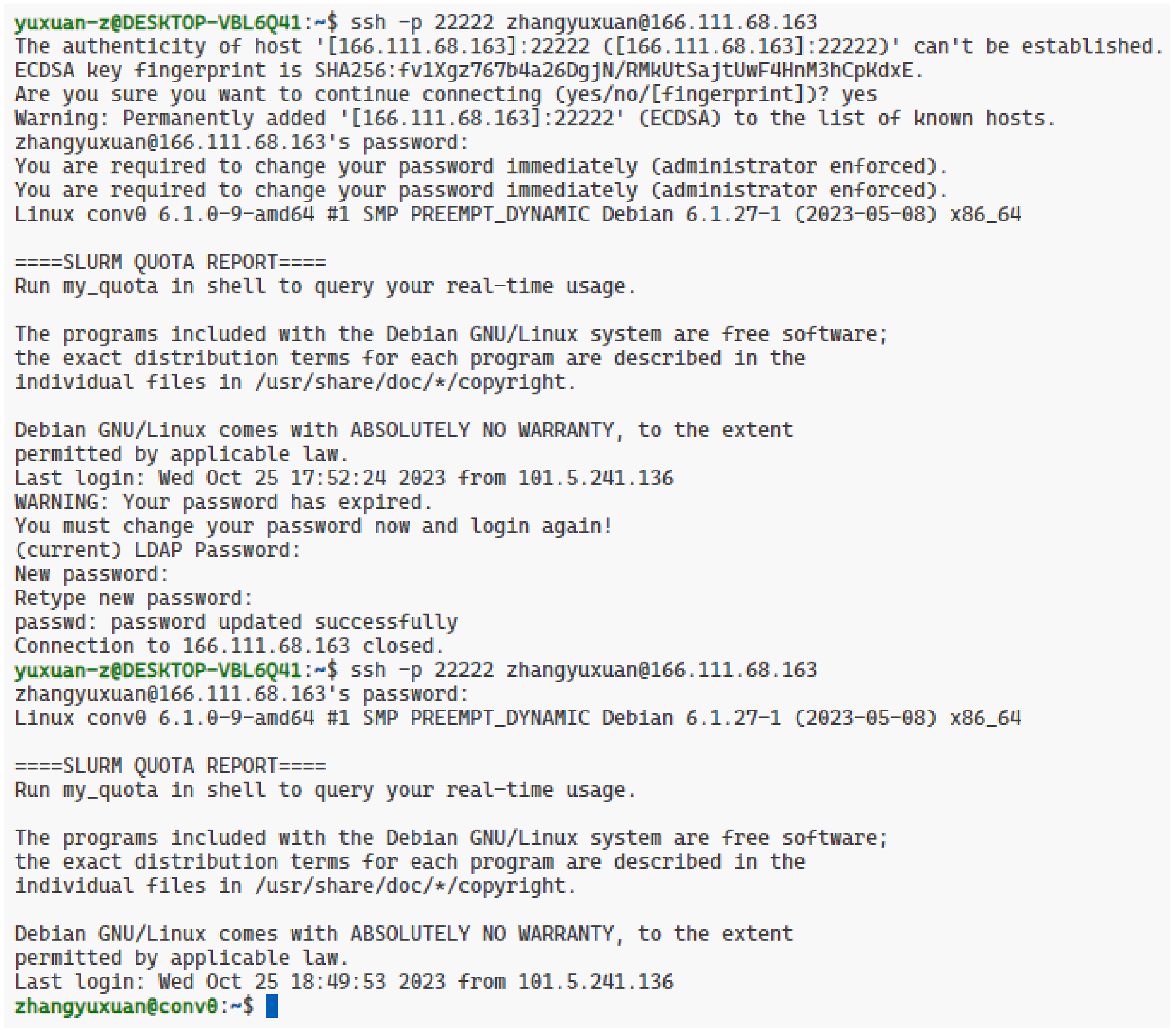
- 166.111.68.163 是集群的IP地址
- -p 22222 指定通过开放端口22222登录
- 用户名和密码通过网络学堂作业发放
- Linux下密码输入是不会回显的

首次登陆需要更改初始密码为新密码（>=12位）



```
Last login: Wed Oct 25 17:52:24 2023 from 101.5.241.136
WARNING: Your password has expired.
You must change your password now and login again!
(current) LDAP Password:
New password:
Retype new password:
passwd: password updated successfully
Connection to 166.111.68.163 closed.
```

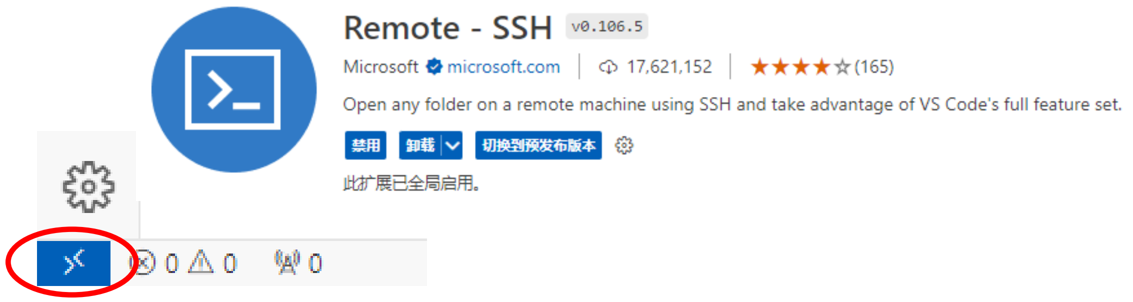
修改后会强制登出，再次通过ssh登陆即可



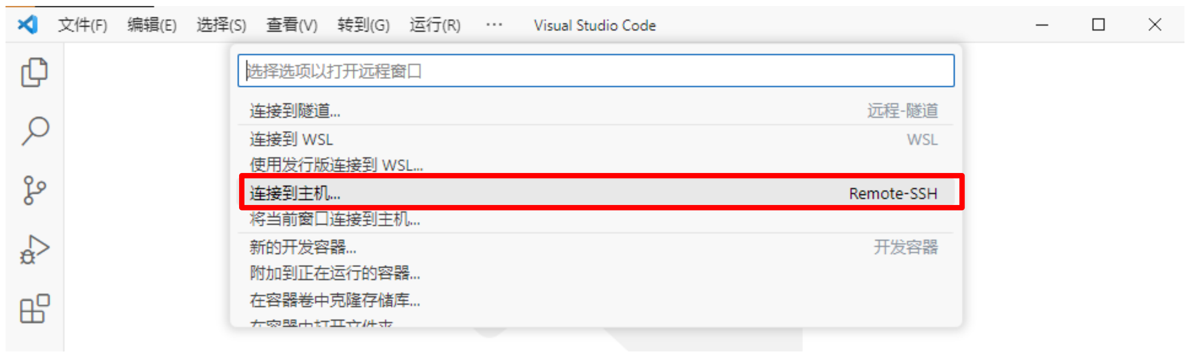
```
yuxuan-z@DESKTOP-VBL6Q41:~$ ssh -p 22222 zhangyuxuan@166.111.68.163
The authenticity of host '[166.111.68.163]:22222 ([166.111.68.163]:22222)' can't be established.
ECDSA key fingerprint is SHA256:fv1Xgz767b4a26DgjN/RMkUtSajtUwF4HnM3hCpKdxE.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[166.111.68.163]:22222' (ECDSA) to the list of known hosts.
zhangyuxuan@166.111.68.163's password:
You are required to change your password immediately (administrator enforced).
You are required to change your password immediately (administrator enforced).
Linux conv0 6.1.0-9-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.27-1 (2023-05-08) x86_64

====SLURM QUOTA REPORT====
Run my_quota in shell to query your real-time usage.

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Oct 25 17:52:24 2023 from 101.5.241.136
WARNING: Your password has expired.
You must change your password now and login again!
(current) LDAP Password:
New password:
Retype new password:
passwd: password updated successfully
Connection to 166.111.68.163 closed.
yuxuan-z@DESKTOP-VBL6Q41:~$ ssh -p 22222 zhangyuxuan@166.111.68.163
zhangyuxuan@166.111.68.163's password:
Linux conv0 6.1.0-9-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.27-1 (2023-05-08) x86_64

====SLURM QUOTA REPORT====
Run my_quota in shell to query your real-time usage.

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Oct 25 18:49:53 2023 from 101.5.241.136
zhangyuxuan@conv0:~$
```

VS Code下的SSH (1)

安装Remote-SSH插件（更推荐Remote Development插件包）
安装后VS Code的左下角会出现Remote Connection的按钮 →


Remote - SSH v0.106.5
Microsoft microsoft.com | 17,621,152 | ★★★★☆ (165)
Open any folder on a remote machine using SSH and take advantage of VS Code's full feature set.
禁用 卸载 切换到预发布版本
此扩展已全局启用。

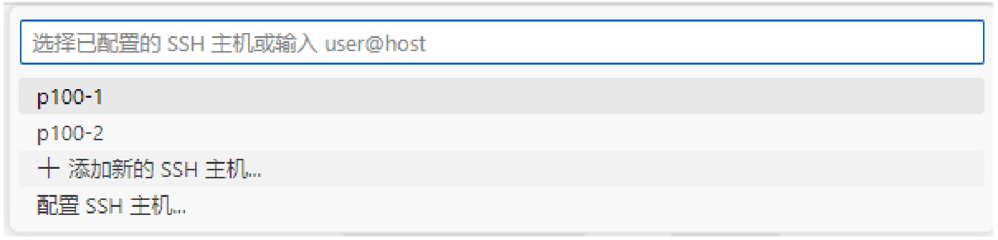
⊗ 0 ⚠ 0 0


文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) ··· Visual Studio Code

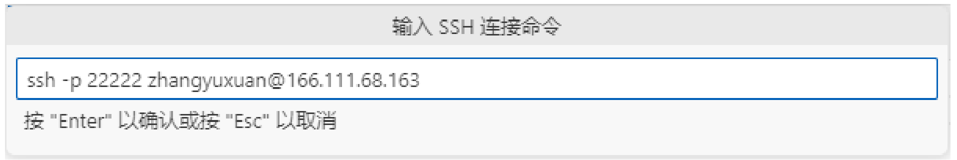
选择选项以打开远程窗口

连接到隧道...	远程-隧道
连接到 WSL	WSL
使用发行版连接到 WSL...	
连接到主机...	Remote-SSH
将当前窗口连接到主机...	
新的开发容器...	开发容器
附加到正在运行的容器...	
在容器卷中克隆存储库...	

点击后，会出现对话窗，选择 **连接到主机… Remote-SSH**
初始时点击 **添加新的SSH主机**，在后续可以直接点击连接


选择已配置的 SSH 主机或输入 user@host
p100-1
p100-2
+ 添加新的 SSH 主机...
配置 SSH 主机...

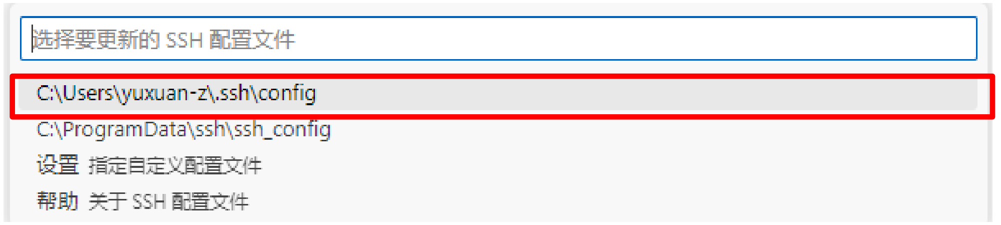
点击后会要求输入SSH连接命令，即前一页的ssh命令


输入 SSH 连接命令

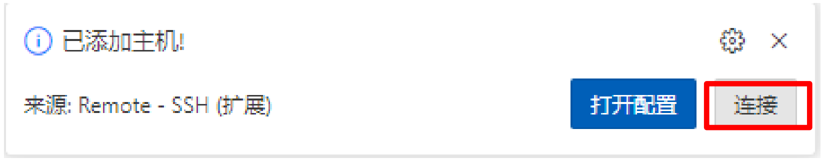
```
ssh -p 22222 zhangyuxuan@166.111.68.163
```

按 "Enter" 以确认或按 "Esc" 以取消

并写入SSH配置文件，默认为~/.ssh/config

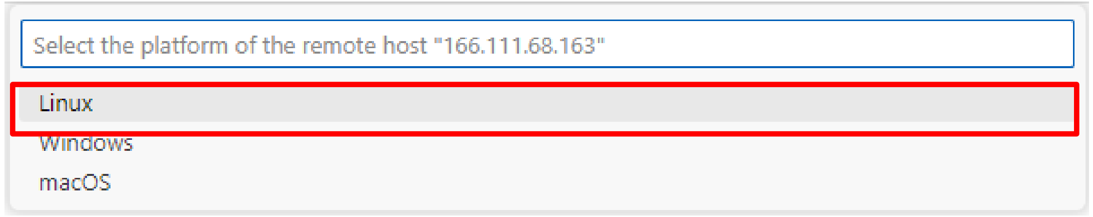

选择要更新的 SSH 配置文件
C:\Users\yuxuan-z\.ssh\config
C:\ProgramData\ssh\ssh_config
设置 指定自定义配置文件
帮助 关于 SSH 配置文件

提示配置成功，点击 **连接**


已添加主机!
来源: Remote - SSH (扩展)
打开配置 连接

VS Code下的SSH (2)

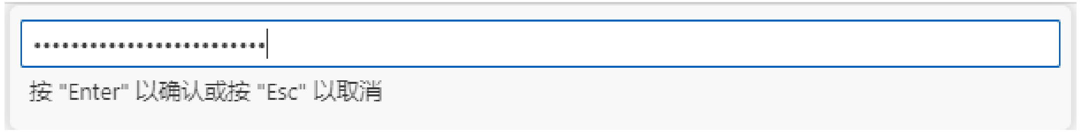
点击连接后，会创建新的工作窗口，指定远程目标为Linux



Select the platform of the remote host "166.111.68.163"

Linux
Windows
macOS

之后要求输入密码



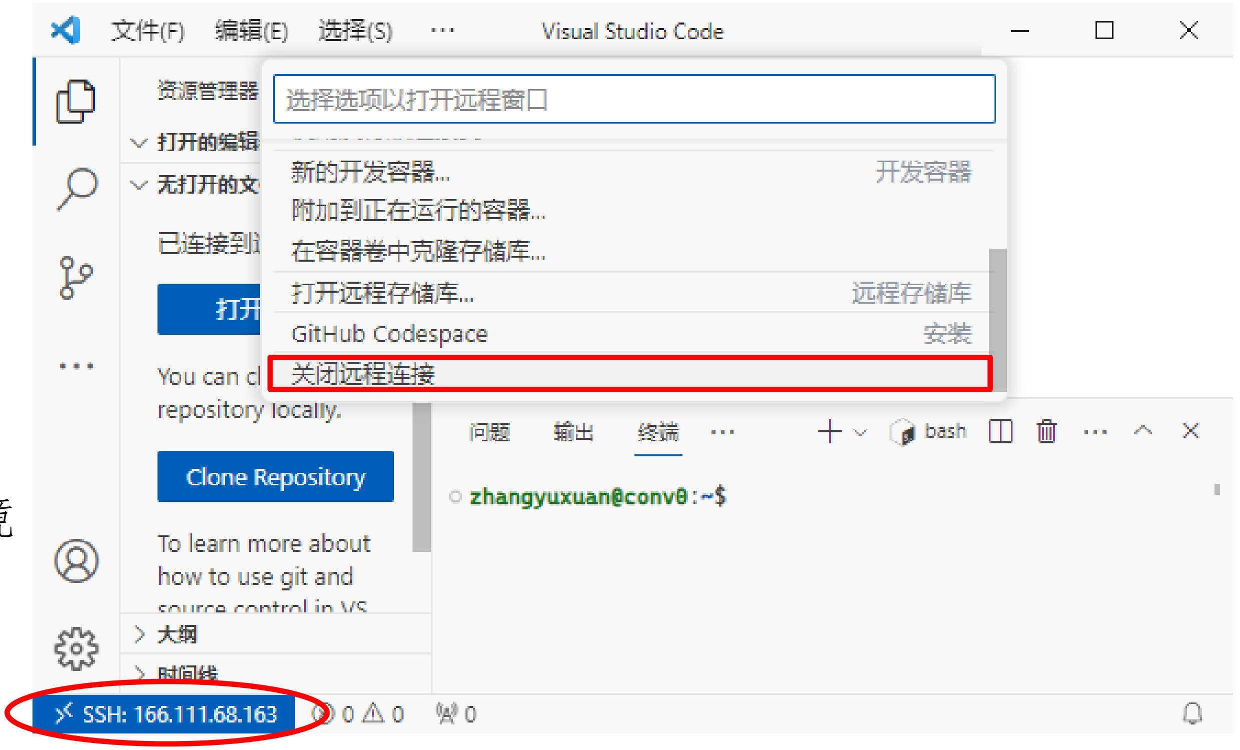
```
··························
```

按 "Enter" 以确认或按 "Esc" 以取消

登录成功后，左下角会显示 **SSH:连接名**
接下来可以像平时用VS Code那样打开文件夹进入工作环境
注意VS Code中大部分插件需要在远程环境中再次安装

单击右下角连接，在对话窗中选择最后一项
关闭远程连接 即可登出服务器并关闭当前窗口

重新启动窗口，点击 166.111.68.163 登录

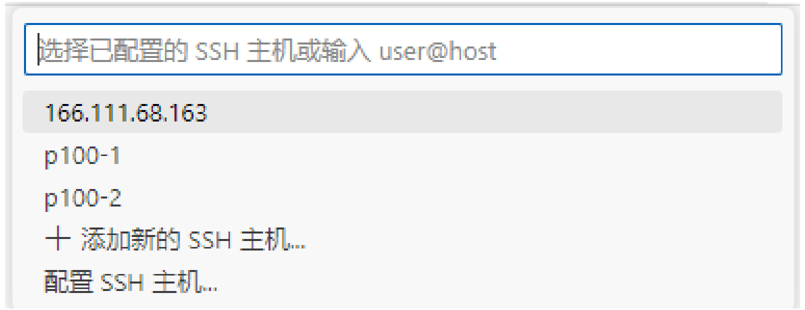


选择选项以打开远程窗口

新的开发容器...	开发容器
附加到正在运行的容器...	
在容器卷中克隆存储库...	
打开远程存储库...	远程存储库
GitHub Codespace	安装
关闭远程连接	

```
zhangyuxuan@conv0:~$
```

SSH: 166.111.68.163



选择已配置的 SSH 主机或输入 user@host

166.111.68.163
p100-1
p100-2
+ 添加新的 SSH 主机...
配置 SSH 主机...

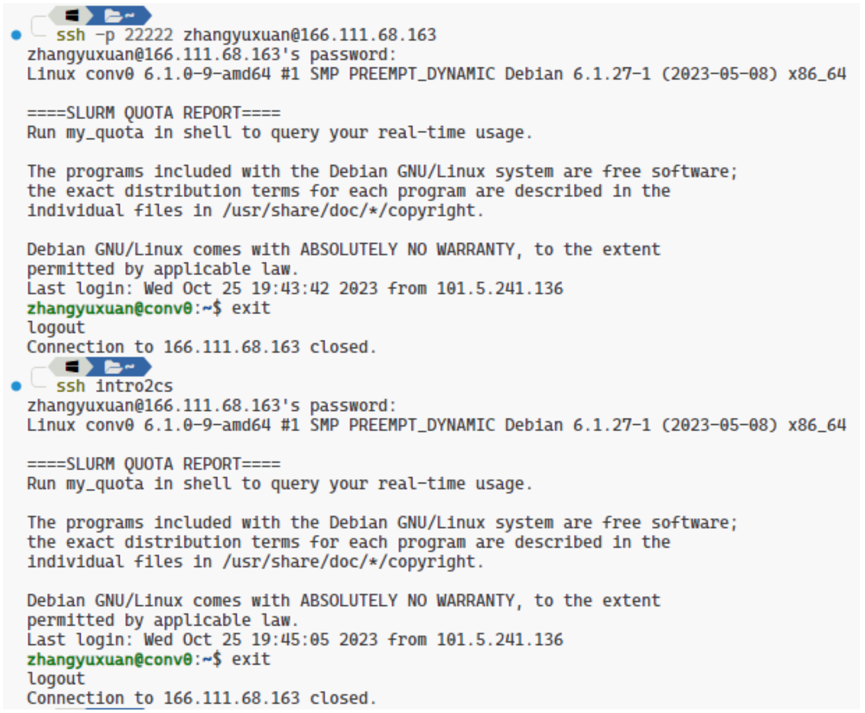
SSH进阶用法(1)

每次输入用户名和IP地址、指定端口号很麻烦？

可以修改~/.ssh/config文件中这个连接的名称 Host

再次登陆时，直接 ssh **连接名称** 就行

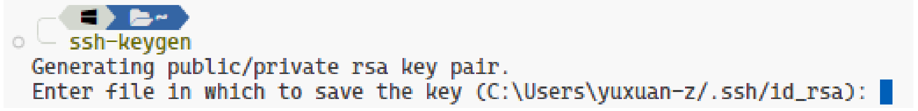
```
# ~/.ssh/config

# Host 166.111.68.163
Host intro2cs
  HostName 166.111.68.163
  Port 22222
  User zhangyuxuan
```



```
ssh -p 22222 zhangyuxuan@166.111.68.163
zhangyuxuan@166.111.68.163's password:
Linux conv0 6.1.0-9-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.27-1 (2023-05-08) x86_64

====SLURM QUOTA REPORT====
Run my_quota in shell to query your real-time usage.

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Oct 25 19:43:42 2023 from 101.5.241.136
zhangyuxuan@conv0:~$ exit
logout
Connection to 166.111.68.163 closed.
ssh intro2cs
zhangyuxuan@166.111.68.163's password:
Linux conv0 6.1.0-9-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.27-1 (2023-05-08) x86_64

====SLURM QUOTA REPORT====
Run my_quota in shell to query your real-time usage.

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Oct 25 19:45:05 2023 from 101.5.241.136
zhangyuxuan@conv0:~$ exit
logout
Connection to 166.111.68.163 closed.
```

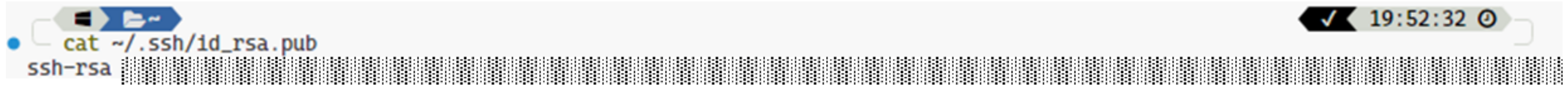
每次登录、切换都需要输入密码很麻烦？

在本地通过ssh-keygen命令生成SSH密钥对，一直按Enter键用默认的就行



```
ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (C:\Users\yuxuan-z/.ssh/id_rsa):
```

生成后通过cat命令查看~/.ssh/id_rsa.pub的内容，选中复制



```
cat ~/.ssh/id_rsa.pub                                          ✓  19:52:32
ssh-rsa [redacted]
```

登录远程，在自己的家目录~下创建.ssh文件夹

```
zhangyuxuan@conv0:~$ mkdir ~/.ssh
```

通过echo命令将复制的内容（鼠标右击粘贴）写入.ssh/authorized_keys文件中

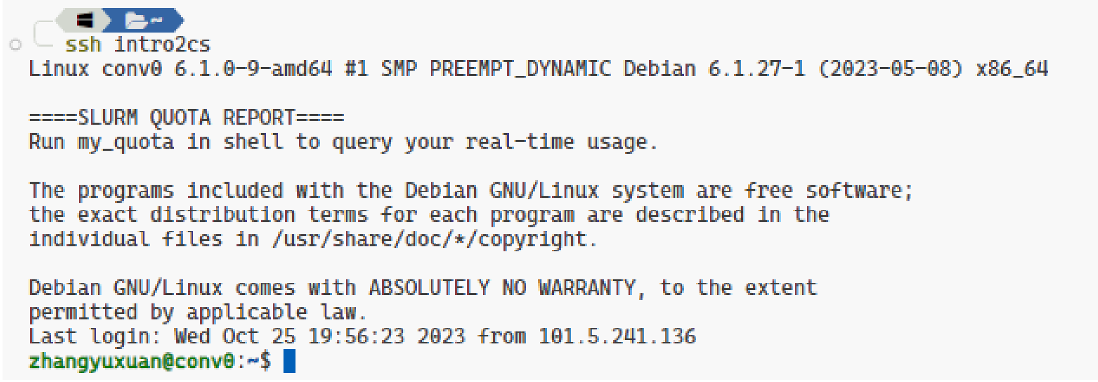
echo id_rsa.pub的内容 >> ~/.ssh/authorized_keys

用chmod命令将.ssh文件夹设置权限700

而.ssh/authorized_keys的权限为600

```
zhangyuxuan@conv0:~$ chmod 700 ~/.ssh && chmod 600 ~/.ssh/authorized_keys
```

登出再重新登录，之后都不再需要输入密码了！



```
ssh intro2cs
Linux conv0 6.1.0-9-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.27-1 (2023-05-08) x86_64

====SLURM QUOTA REPORT====
Run my_quota in shell to query your real-time usage.

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Oct 25 19:56:23 2023 from 101.5.241.136
zhangyuxuan@conv0:~$
```

SSH进阶用法(2)

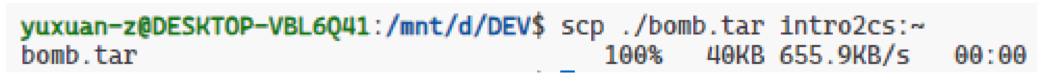
VS Code下载和上传文件很慢，有没有更快的方法？

建议使用scp命令来实现文件传输

具体可参考文档 https://www.runoob.com/linux/linux-comm-scp.html

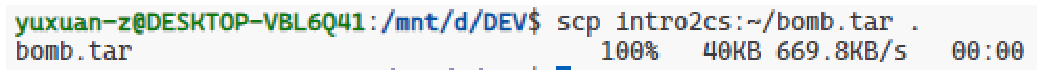
注意scp命令必须是在本地执行

1. 从本地上传文件到服务器



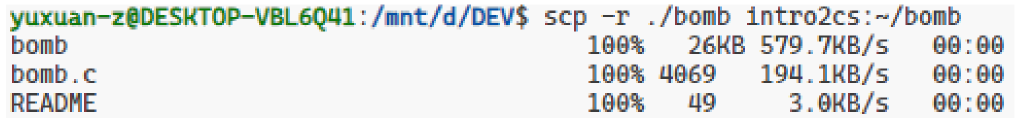
```
yuxuan-z@DESKTOP-VBL6Q41:/mnt/d/DEV$ scp ./bomb.tar intro2cs:~
bomb.tar                                    100%   40KB 655.9KB/s   00:00
```

2. 从服务器下载文件



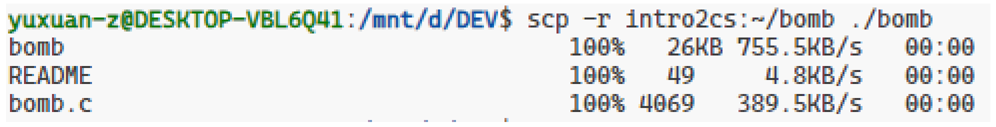
```
yuxuan-z@DESKTOP-VBL6Q41:/mnt/d/DEV$ scp intro2cs:~/bomb.tar .
bomb.tar                                    100%   40KB 669.8KB/s   00:00
```

3. 从本地上传文件夹到服务器



```
yuxuan-z@DESKTOP-VBL6Q41:/mnt/d/DEV$ scp -r ./bomb intro2cs:~/bomb
bomb                                        100%   26KB 579.7KB/s   00:00
bomb.c                                      100% 4069   194.1KB/s   00:00
README                                      100%   49     3.0KB/s   00:00
```

4. 从服务器下载文件夹到本地



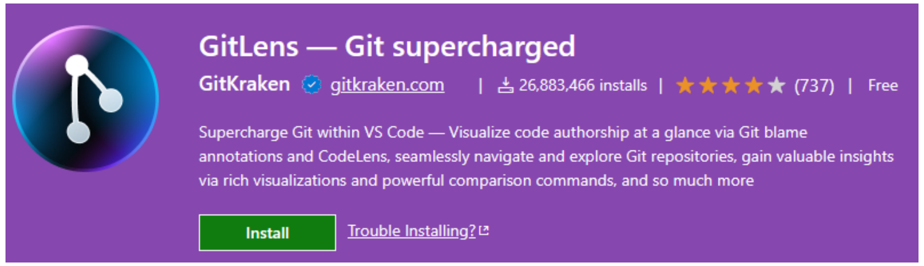
```
yuxuan-z@DESKTOP-VBL6Q41:/mnt/d/DEV$ scp -r intro2cs:~/bomb ./bomb
bomb                                        100%   26KB 755.5KB/s   00:00
README                                      100%   49     4.8KB/s   00:00
bomb.c                                      100% 4069   389.5KB/s   00:00
```

*** 如何备份/管理实验中的代码？**

一般会使用Git来管理代码的版本
同时基于 GitHub 或 Tsinghua GitLab

Git为《软件工程》课的内容，这里不做展开
有兴趣的同学可参考 2023酒井科协暑培回放
https://www.bilibili.com/video/BV1DN411m74Q

VS Code下推荐的插件为 GitLens



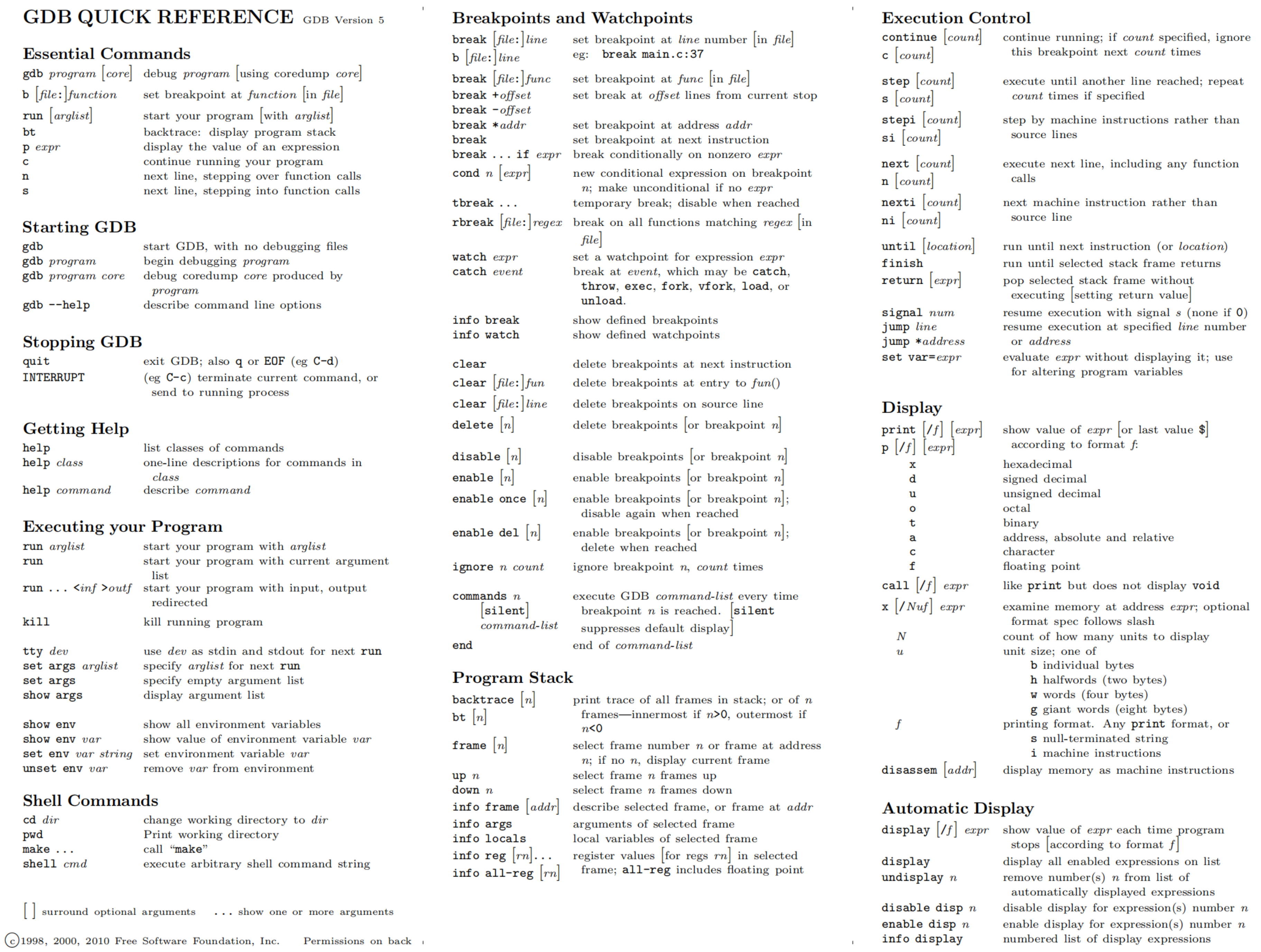
GitLens — Git supercharged

GitKraken gitkraken.com | 26,883,466 installs | ★★★★★ (737) | Free

Supercharge Git within VS Code — Visualize code authorship at a glance via Git blame annotations and CodeLens, seamlessly navigate and explore Git repositories, gain valuable insights via rich visualizations and powerful comparison commands, and so much more

GDB手册(0)

Princeton University
COS 432 Information Security



GDB QUICK REFERENCE GDB Version 5

Essential Commands

gdb *program* [*core*]	debug *program* [using coredump *core*]
b [*file*:]*function*	set breakpoint at *function* [in *file*]
run [*arglist*]	start your program [with *arglist*]
bt	backtrace: display program stack
p *expr*	display the value of an expression
c	continue running your program
n	next line, stepping over function calls
s	next line, stepping into function calls

Starting GDB

gdb	start GDB, with no debugging files
gdb *program*	begin debugging *program*
gdb *program core*	debug coredump *core* produced by *program*
gdb --help	describe command line options

Stopping GDB

quit	exit GDB; also q or EOF (eg C-d)
INTERRUPT	(eg C-c) terminate current command, or send to running process

Getting Help

help	list classes of commands
help *class*	one-line descriptions for commands in *class*
help *command*	describe *command*

Executing your Program

run *arglist*	start your program with *arglist*
run	start your program with current argument list
run ... <*inf* >*outf*	start your program with input, output redirected
kill	kill running program
tty *dev*	use *dev* as stdin and stdout for next run
set args *arglist*	specify *arglist* for next run
set args	specify empty argument list
show args	display argument list
show env	show all environment variables
show env *var*	show value of environment variable *var*
set env *var string*	set environment variable *var*
unset env *var*	remove *var* from environment

Shell Commands

cd *dir*	change working directory to *dir*
pwd	Print working directory
make ...	call "make"
shell *cmd*	execute arbitrary shell command string

[ ] surround optional arguments ... show one or more arguments

Breakpoints and Watchpoints

break [*file*:]*line* b [*file*:]*line*	set breakpoint at *line* number [in *file*] eg: break main.c:37
break [*file*:]*func*	set breakpoint at *func* [in *file*]

break +*offset*
break -*offset*
	set break at *offset* lines from current stop

break **addr*	set breakpoint at address *addr*
break	set breakpoint at next instruction
break ... if *expr*	break conditionally on nonzero *expr*
cond *n* [*expr*]	new conditional expression on breakpoint *n*; make unconditional if no *expr*
tbreak ...	temporary break; disable when reached
rbreak [*file*:]*regex*	break on all functions matching *regex* [in *file*]
watch *expr*	set a watchpoint for expression *expr*
catch *event*	break at *event*, which may be catch, throw, exec, fork, vfork, load, or unload.
info break	show defined breakpoints
info watch	show defined watchpoints
clear	delete breakpoints at next instruction
clear [*file*:]*fun*	delete breakpoints at entry to *fun*()
clear [*file*:]*line*	delete breakpoints on source line
delete [*n*]	delete breakpoints [or breakpoint *n*]
disable [*n*]	disable breakpoints [or breakpoint *n*]
enable [*n*]	enable breakpoints [or breakpoint *n*]
enable once [*n*]	enable breakpoints [or breakpoint *n*]; disable again when reached
enable del [*n*]	enable breakpoints [or breakpoint *n*]; delete when reached
ignore *n count*	ignore breakpoint *n*, *count* times
commands *n* [silent] *command-list*	execute GDB *command-list* every time breakpoint *n* is reached. [silent suppresses default display]
end	end of *command-list*

Program Stack

backtrace [*n*] bt [*n*]	print trace of all frames in stack; or of *n* frames—innermost if *n*>0, outermost if *n*<0
frame [*n*]	select frame number *n* or frame at address *n*; if no *n*, display current frame
up *n*	select frame *n* frames up
down *n*	select frame *n* frames down
info frame [*addr*]	describe selected frame, or frame at *addr*
info args	arguments of selected frame
info locals	local variables of selected frame
info reg [*rn*]...	register values [for regs *rn*] in selected frame; all-reg includes floating point
info all-reg [*rn*]	

Execution Control

continue [*count*] c [*count*]	continue running; if *count* specified, ignore this breakpoint next *count* times
step [*count*] s [*count*]	execute until another line reached; repeat *count* times if specified
stepi [*count*] si [*count*]	step by machine instructions rather than source lines
next [*count*] n [*count*]	execute next line, including any function calls
nexti [*count*] ni [*count*]	next machine instruction rather than source line
until [*location*]	run until next instruction (or *location*)
finish	run until selected stack frame returns
return [*expr*]	pop selected stack frame without executing [setting return value]
signal *num*	resume execution with signal *s* (none if 0)
jump *line* jump **address*	resume execution at specified *line* number or *address*
set var=*expr*	evaluate *expr* without displaying it; use for altering program variables

Display

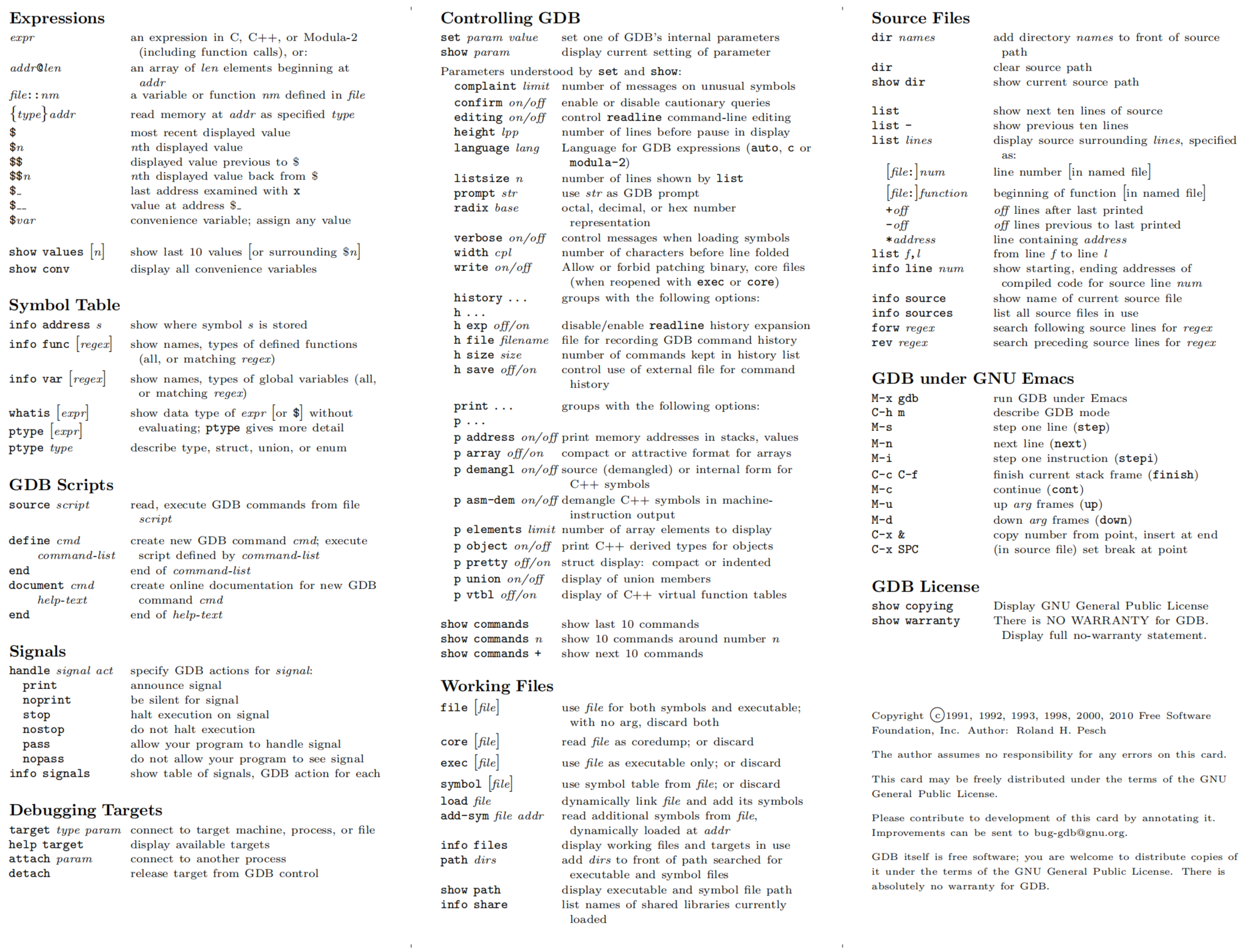
print [/*f*] [*expr*] p [/*f*] [*expr*]	show value of *expr* [or last value $] according to format *f*:
x	hexadecimal
d	signed decimal
u	unsigned decimal
o	octal
t	binary
a	address, absolute and relative
c	character
f	floating point
call [/*f*] *expr*	like print but does not display void
x [/*Nuf*] *expr*	examine memory at address *expr*; optional format spec follows slash
N	count of how many units to display
u	unit size; one of b individual bytes h halfwords (two bytes) w words (four bytes) g giant words (eight bytes)
f	printing format. Any print format, or s null-terminated string i machine instructions
disassem [*addr*]	display memory as machine instructions

Automatic Display

display [/*f*] *expr*	show value of *expr* each time program stops [according to format *f*]
display	display all enabled expressions on list
undisplay *n*	remove number(s) *n* from list of automatically displayed expressions
disable disp *n*	disable display for expression(s) number *n*
enable disp *n*	enable display for expression(s) number *n*
info display	numbered list of display expressions

GDB手册(1)

Princeton University
COS 432 Information Security



Expressions

expr	an expression in C, C++, or Modula-2 (including function calls), or:
addr@*len*	an array of *len* elements beginning at *addr*
file::*nm*	a variable or function *nm* defined in *file*
{*type*}*addr*	read memory at *addr* as specified *type*
`$`	most recent displayed value
`$`*n*	*n*th displayed value
`$$`	displayed value previous to $
`$$`*n*	*n*th displayed value back from $
`$_`	last address examined with `x`
`$__`	value at address $_
`$`*var*	convenience variable; assign any value
`show values` [*n*]	show last 10 values [or surrounding $*n*]
`show conv`	display all convenience variables

Symbol Table

`info address` *s*	show where symbol *s* is stored
`info func` [*regex*]	show names, types of defined functions (all, or matching *regex*)
`info var` [*regex*]	show names, types of global variables (all, or matching *regex*)
`whatis` [*expr*]	show data type of *expr* [or `$`] without evaluating; `ptype` gives more detail
`ptype` [*expr*]	
`ptype` *type*	describe type, struct, union, or enum

GDB Scripts

`source` *script*	read, execute GDB commands from file *script*
`define` *cmd* *command-list*	create new GDB command *cmd*; execute script defined by *command-list*
`end`	end of *command-list*
`document` *cmd* *help-text*	create online documentation for new GDB command *cmd*
`end`	end of *help-text*

Signals

`handle` *signal act*	specify GDB actions for *signal*:
`print`	announce signal
`noprint`	be silent for signal
`stop`	halt execution on signal
`nostop`	do not halt execution
`pass`	allow your program to handle signal
`nopass`	do not allow your program to see signal
`info signals`	show table of signals, GDB action for each

Debugging Targets

`target` *type param*	connect to target machine, process, or file
`help target`	display available targets
`attach` *param*	connect to another process
`detach`	release target from GDB control

Controlling GDB

`set` *param value*	set one of GDB's internal parameters
`show` *param*	display current setting of parameter

Parameters understood by `set` and `show`:

`complaint` *limit*	number of messages on unusual symbols
`confirm` *on/off*	enable or disable cautionary queries
`editing` *on/off*	control `readline` command-line editing
`height` *lpp*	number of lines before pause in display
`language` *lang*	Language for GDB expressions (`auto`, `c` or `modula-2`)
`listsize` *n*	number of lines shown by `list`
`prompt` *str*	use *str* as GDB prompt
`radix` *base*	octal, decimal, or hex number representation
`verbose` *on/off*	control messages when loading symbols
`width` *cpl*	number of characters before line folded
`write` *on/off*	Allow or forbid patching binary, core files (when reopened with `exec` or `core`)
`history ...` / `h ...`	groups with the following options:
`h exp` *off/on*	disable/enable `readline` history expansion
`h file` *filename*	file for recording GDB command history
`h size` *size*	number of commands kept in history list
`h save` *off/on*	control use of external file for command history
`print ...` / `p ...`	groups with the following options:
`p address` *on/off*	print memory addresses in stacks, values
`p array` *off/on*	compact or attractive format for arrays
`p demangl` *on/off*	source (demangled) or internal form for C++ symbols
`p asm-dem` *on/off*	demangle C++ symbols in machine-instruction output
`p elements` *limit*	number of array elements to display
`p object` *on/off*	print C++ derived types for objects
`p pretty` *off/on*	struct display: compact or indented
`p union` *on/off*	display of union members
`p vtbl` *off/on*	display of C++ virtual function tables
`show commands`	show last 10 commands
`show commands` *n*	show 10 commands around number *n*
`show commands +`	show next 10 commands

Working Files

`file` [*file*]	use *file* for both symbols and executable; with no arg, discard both
`core` [*file*]	read *file* as coredump; or discard
`exec` [*file*]	use *file* as executable only; or discard
`symbol` [*file*]	use symbol table from *file*; or discard
`load` *file*	dynamically link *file* and add its symbols
`add-sym` *file addr*	read additional symbols from *file*, dynamically loaded at *addr*
`info files`	display working files and targets in use
`path` *dirs*	add *dirs* to front of path searched for executable and symbol files
`show path`	display executable and symbol file path
`info share`	list names of shared libraries currently loaded

Source Files

`dir` *names*	add directory *names* to front of source path
`dir`	clear source path
`show dir`	show current source path
`list`	show next ten lines of source
`list -`	show previous ten lines
`list` *lines*	display source surrounding *lines*, specified as:
[*file*:]*num*	line number [in named file]
[*file*:]*function*	beginning of function [in named file]
`+`*off*	*off* lines after last printed
`-`*off*	*off* lines previous to last printed
`*`*address*	line containing *address*
`list` *f*,*l*	from line *f* to line *l*
`info line` *num*	show starting, ending addresses of compiled code for source line *num*
`info source`	show name of current source file
`info sources`	list all source files in use
`forw` *regex*	search following source lines for *regex*
`rev` *regex*	search preceding source lines for *regex*

GDB under GNU Emacs

`M-x gdb`	run GDB under Emacs
`C-h m`	describe GDB mode
`M-s`	step one line (`step`)
`M-n`	next line (`next`)
`M-i`	step one instruction (`stepi`)
`C-c C-f`	finish current stack frame (`finish`)
`M-c`	continue (`cont`)
`M-u`	up *arg* frames (`up`)
`M-d`	down *arg* frames (`down`)
`C-x &`	copy number from point, insert at end
`C-x SPC`	(in source file) set break at point

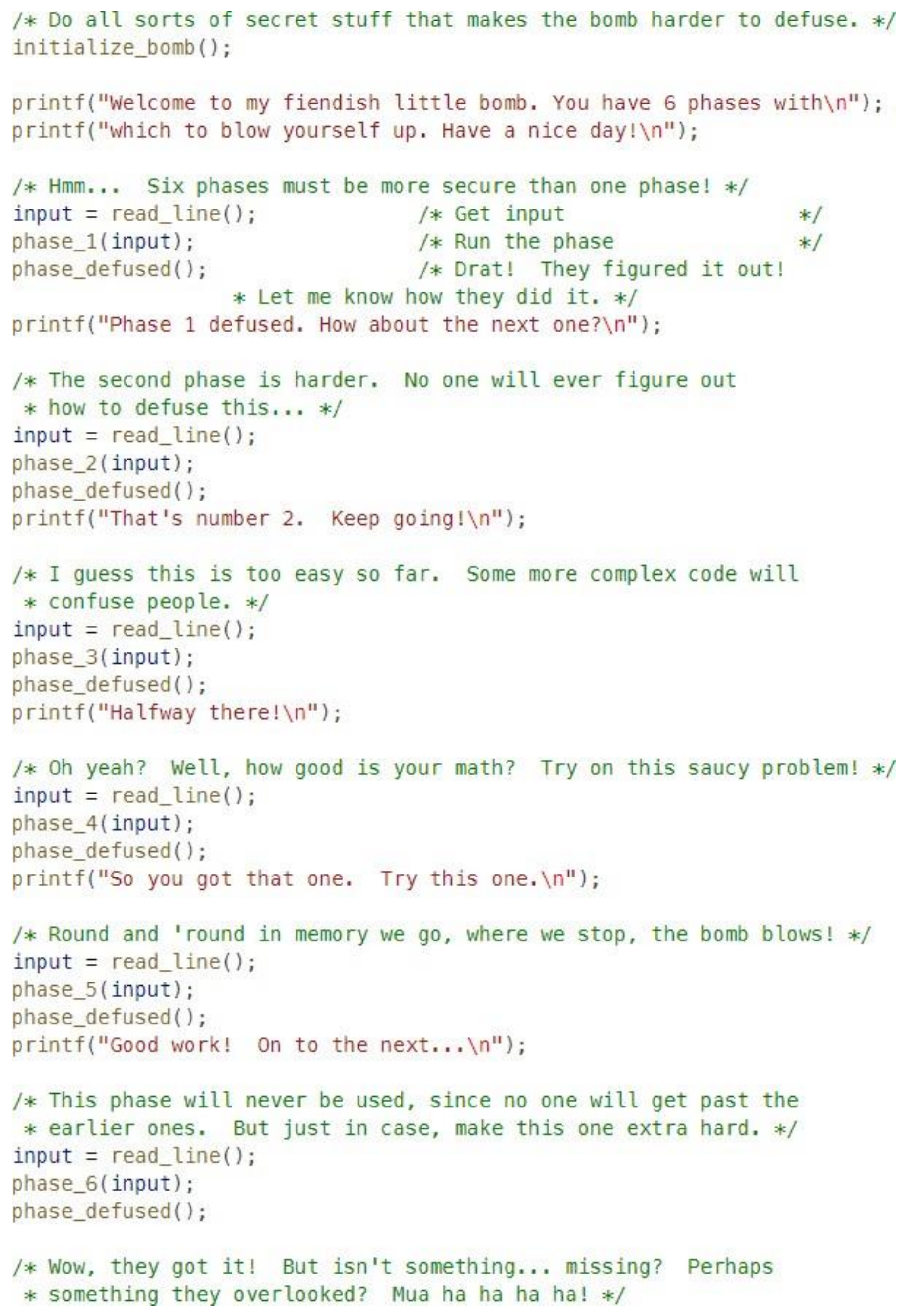
GDB License

`show copying`	Display GNU General Public License
`show warranty`	There is NO WARRANTY for GDB. Display full no-warranty statement.

Bomb Lab 概述

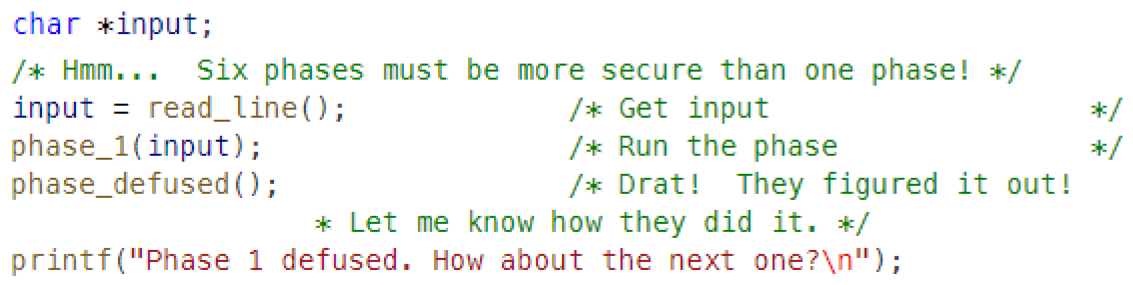
- 任务描述：
  - 程序读取用户输入，并判断是否与“预期答案”一致，如果不一致就会引爆炸弹
  - 可以看到一共有6+1个关卡（+1为隐藏关卡，不作要求）
  - 你需要对 bomb 进行反汇编，阅读汇编代码，并使用gdb进行调试，破解每一关的“预期答案”

- 前置知识：
  - X86-64汇编与C语言（见课件）
  - gdb常用技巧（见《实验导引》）
  - 课程服务器登录（见本导引第一部分）



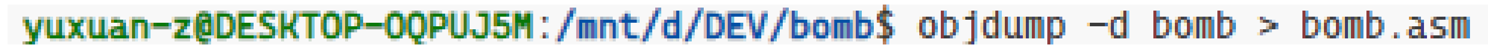
```
/* Do all sorts of secret stuff that makes the bomb harder to defuse. */
initialize_bomb();

printf("Welcome to my fiendish little bomb. You have 6 phases with\n");
printf("which to blow yourself up. Have a nice day!\n");

/* Hmm...  Six phases must be more secure than one phase! */
input = read_line();             /* Get input                   */
phase_1(input);                  /* Run the phase               */
phase_defused();                 /* Drat!  They figured it out!
                  * Let me know how they did it. */
printf("Phase 1 defused. How about the next one?\n");

/* The second phase is harder.  No one will ever figure out
 * how to defuse this... */
input = read_line();
phase_2(input);
phase_defused();
printf("That's number 2.  Keep going!\n");

/* I guess this is too easy so far.  Some more complex code will
 * confuse people. */
input = read_line();
phase_3(input);
phase_defused();
printf("Halfway there!\n");

/* Oh yeah?  Well, how good is your math?  Try on this saucy problem! */
input = read_line();
phase_4(input);
phase_defused();
printf("So you got that one.  Try this one.\n");

/* Round and 'round in memory we go, where we stop, the bomb blows! */
input = read_line();
phase_5(input);
phase_defused();
printf("Good work!  On to the next...\n");

/* This phase will never be used, since no one will get past the
 * earlier ones.  But just in case, make this one extra hard. */
input = read_line();
phase_6(input);
phase_defused();

/* Wow, they got it!  But isn't something... missing?  Perhaps
 * something they overlooked?  Mua ha ha ha ha! */
```

Bomb Lab · Phase1

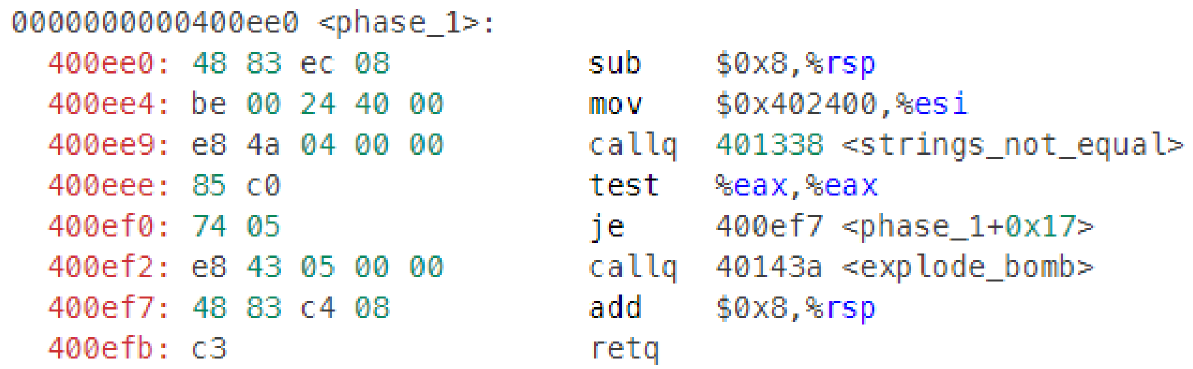


```
char *input;
/* Hmm...  Six phases must be more secure than one phase! */
input = read_line();             /* Get input                   */
phase_1(input);                  /* Run the phase               */
phase_defused();                 /* Drat!  They figured it out!
                      * Let me know how they did it. */
printf("Phase 1 defused. How about the next one?\n");
```

- 不知道该如何下手？看看bomb.c
  还不知道怎么做？看看汇编代码

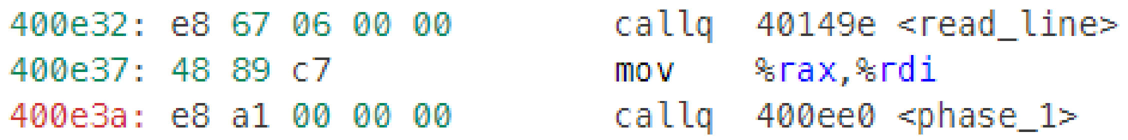


```
yuxuan-z@DESKTOP-OQPUJ5M:/mnt/d/DEV/bomb$ objdump -d bomb > bomb.asm
```



```
0000000000400ee0 <phase_1>:
  400ee0:  48 83 ec 08             sub    $0x8,%rsp
  400ee4:  be 00 24 40 00          mov    $0x402400,%esi
  400ee9:  e8 4a 04 00 00          callq  401338 <strings_not_equal>
  400eee:  85 c0                   test   %eax,%eax
  400ef0:  74 05                   je     400ef7 <phase_1+0x17>
  400ef2:  e8 43 05 00 00          callq  40143a <explode_bomb>
  400ef7:  48 83 c4 08             add    $0x8,%rsp
  400efb:  c3                      retq
```

- phase_1用了strings_not_equal，
  从字面看是判断两个字符串是否完全匹配
  - 调用strings_not_equal之前只对%rsi赋值，没对%rdx及后续参数寄存器赋值
    strings_not_equal只有两个参数

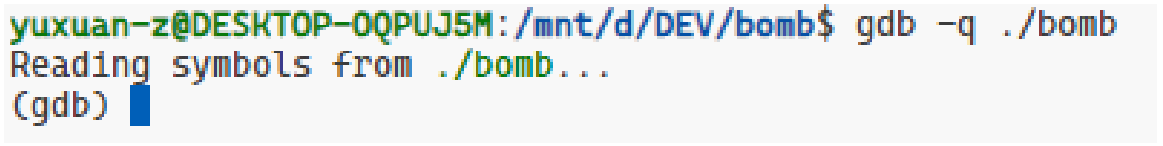


```
400e32:  e8 67 06 00 00          callq  40149e <read_line>
400e37:  48 89 c7                mov    %rax,%rdi
400e3a:  e8 a1 00 00 00          callq  400ee0 <phase_1>
```

  - %rdi的值是啥呢？phase_1被调用前做了什么？
    将input的地址传了进来
  - 大胆猜测，小心求证！%rsi的值就是目标字符串的地址

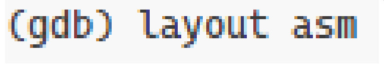
Bomb Lab · Phase1

- 怎么获得(%rsi)的内容呢？使用gdb调试



```
yuxuan-z@DESKTOP-OQPUJ5M:/mnt/d/DEV/bomb$ gdb -q ./bomb
Reading symbols from ./bomb...
(gdb) 
```

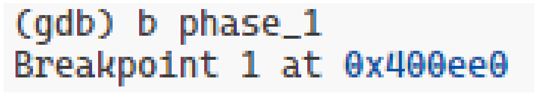
STEP0 怎么实时显示当前执行到的汇编指令呢？



```
(gdb) layout asm
```

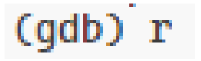
通过Ctrl + L进行刷新

STEP1 怎么停在phase_1呢？



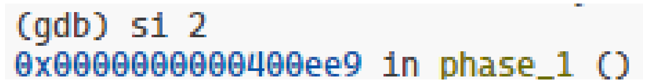
```
(gdb) b phase_1
Breakpoint 1 at 0x400ee0
```

STEP2 怎么启动程序呢？



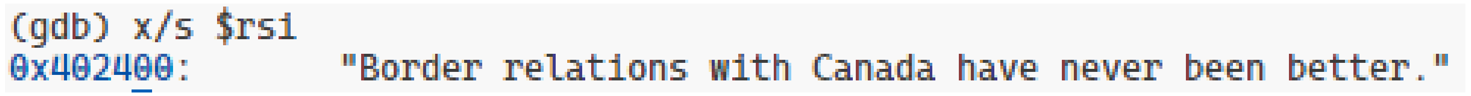
```
(gdb) r
```

STEP3 怎么单步调试（并快进2条指令呢）呢？

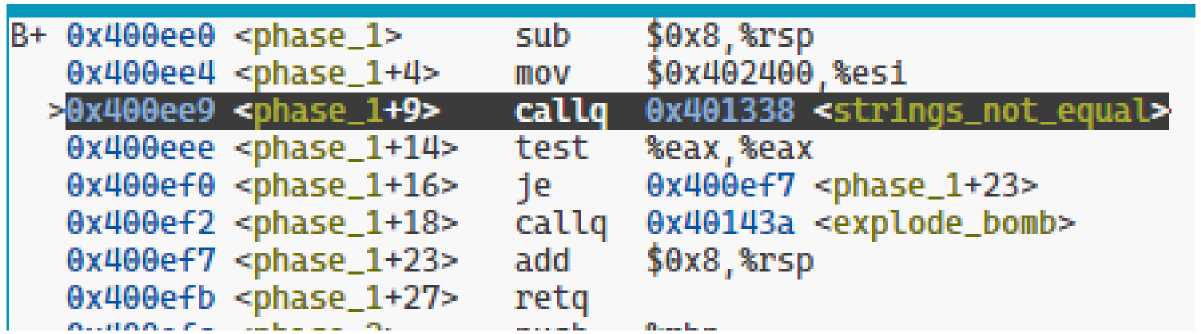


```
(gdb) si 2
0x0000000000400ee9 in phase_1 ()
```

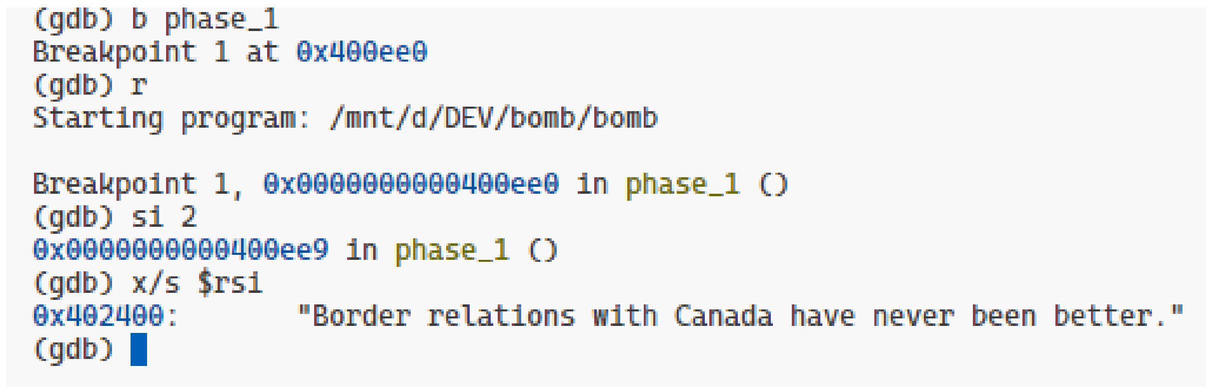
STEP4 怎么输出(%rsi)呢？



```
(gdb) x/s $rsi
0x402400:       "Border relations with Canada have never been better."
```



```
B+ 0x400ee0 <phase_1>       sub    $0x8,%rsp
   0x400ee4 <phase_1+4>     mov    $0x402400,%esi
  >0x400ee9 <phase_1+9>     callq  0x401338 <strings_not_equal>
   0x400eee <phase_1+14>    test   %eax,%eax
   0x400ef0 <phase_1+16>    je     0x400ef7 <phase_1+23>
   0x400ef2 <phase_1+18>    callq  0x40143a <explode_bomb>
   0x400ef7 <phase_1+23>    add    $0x8,%rsp
   0x400efb <phase_1+27>    retq
```



```
(gdb) b phase_1
Breakpoint 1 at 0x400ee0
(gdb) r
Starting program: /mnt/d/DEV/bomb/bomb

Breakpoint 1, 0x0000000000400ee0 in phase_1 ()
(gdb) si 2
0x0000000000400ee9 in phase_1 ()
(gdb) x/s $rsi
0x402400:       "Border relations with Canada have never been better."
(gdb) 
```

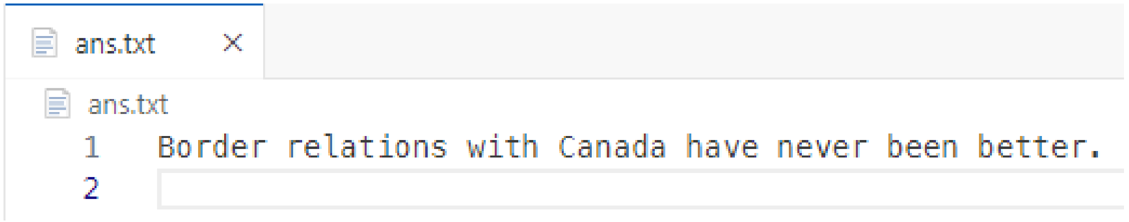
Bomb Lab · Phase1

- 保存(%rsi)的内容，准备进入phase_2

  STEP5 通过echo将答案输出到ans.txt中

  gdb中通过! 表示使用shell来执行该命令

```
(gdb) !echo Border relations with Canada have never been better. >> ans.txt
```



```
ans.txt
ans.txt
Border relations with Canada have never been better.

```

STEP6 删除phase_1的断点，在phase_2打断点

```
(gdb) d breakpoints 1
```

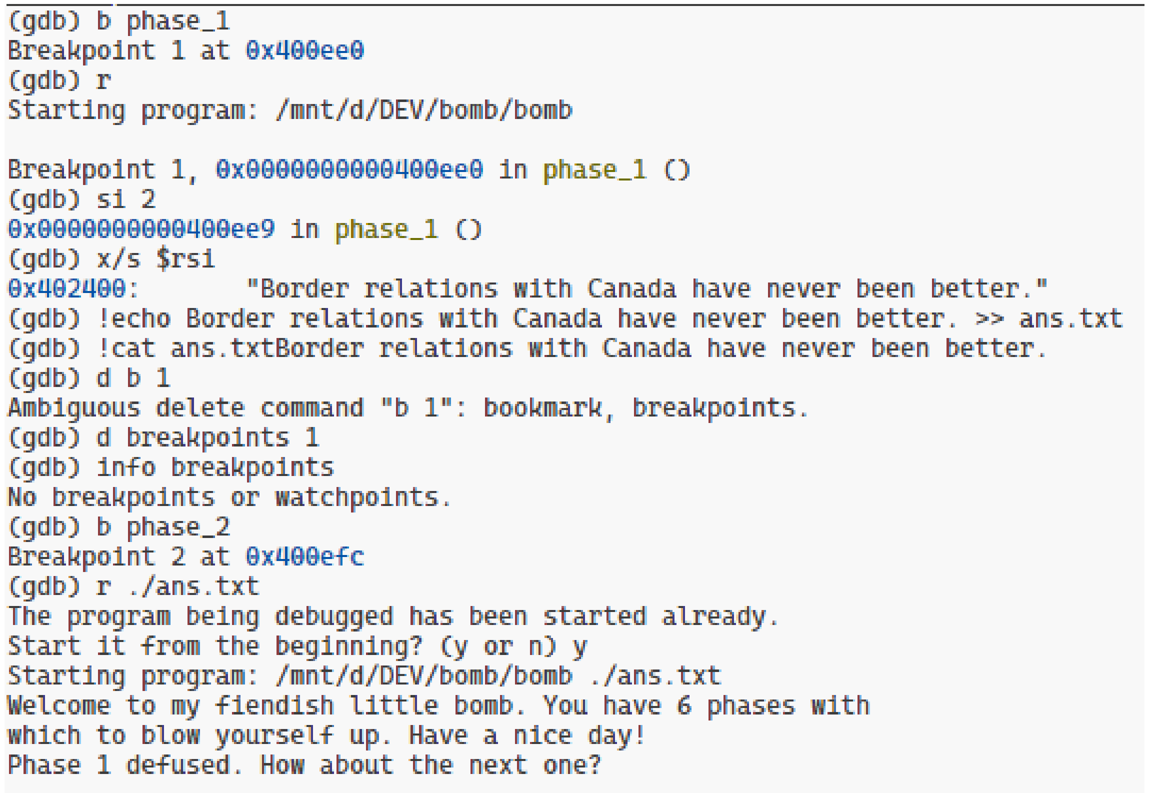
```
(gdb) b phase_2
Breakpoint 2 at 0x400efc
```

可以用tb

STEP7 重新运行，并指定argv[1]为ans.txt

r/run后面可以接argv

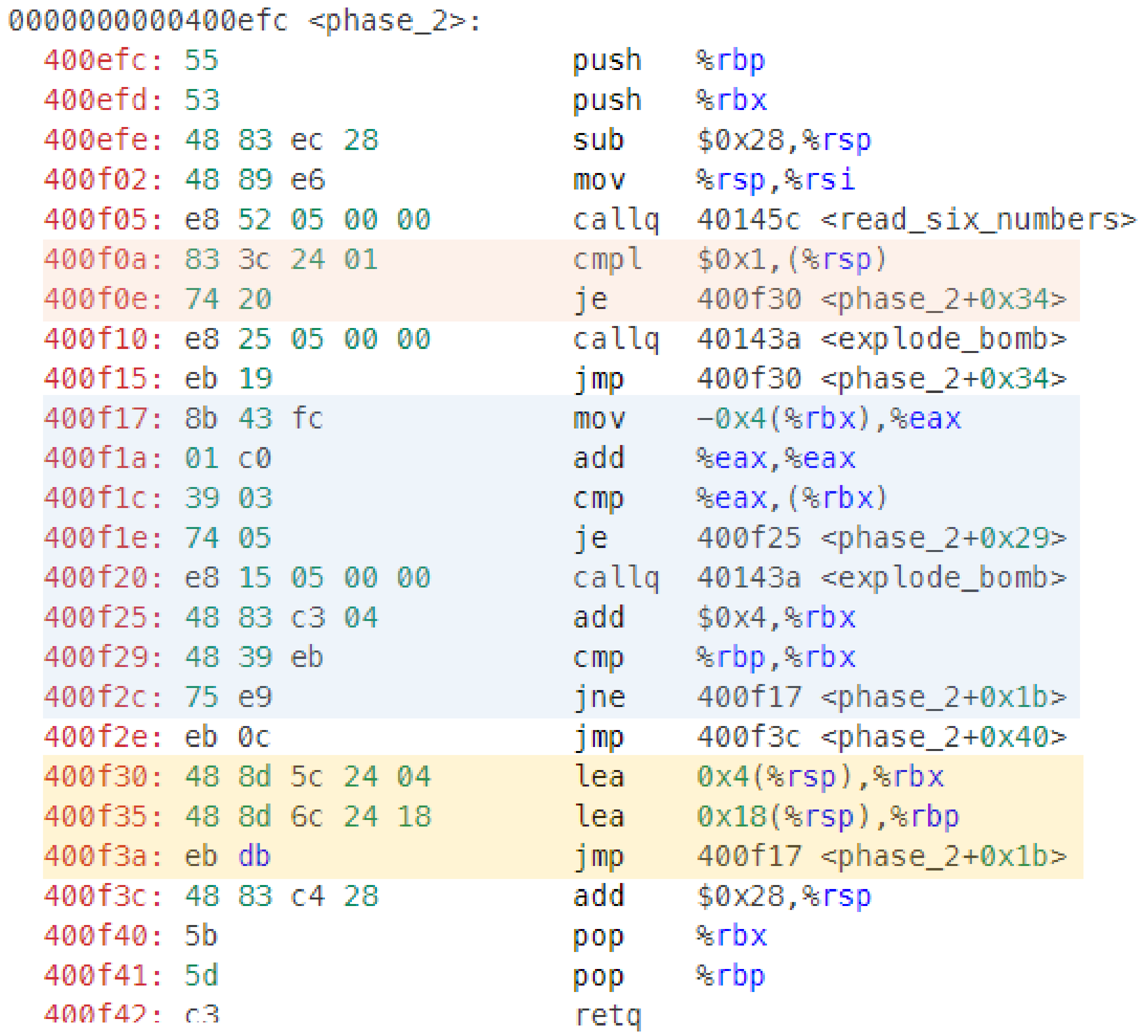
```
(gdb) r ./ans.txt
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /mnt/d/DEV/bomb/bomb ./ans.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
```



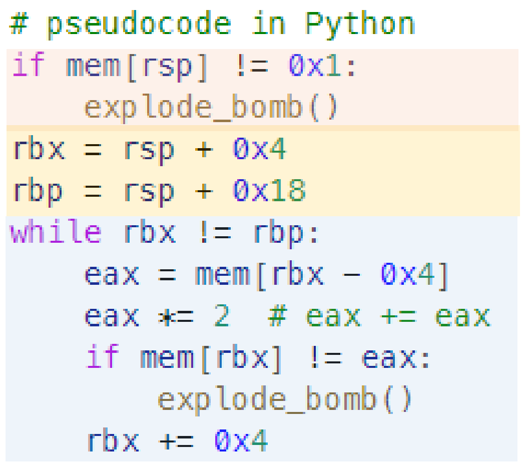
```
(gdb) b phase_1
Breakpoint 1 at 0x400ee0
(gdb) r
Starting program: /mnt/d/DEV/bomb/bomb

Breakpoint 1, 0x0000000000400ee0 in phase_1 ()
(gdb) si 2
0x0000000000400ee9 in phase_1 ()
(gdb) x/s $rsi
0x402400:       "Border relations with Canada have never been better."
(gdb) !echo Border relations with Canada have never been better. >> ans.txt
(gdb) !cat ans.txtBorder relations with Canada have never been better.
(gdb) d b 1
Ambiguous delete command "b 1": bookmark, breakpoints.
(gdb) d breakpoints 1
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb) b phase_2
Breakpoint 2 at 0x400efc
(gdb) r ./ans.txt
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /mnt/d/DEV/bomb/bomb ./ans.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
```

Bomb Lab · Phase 2

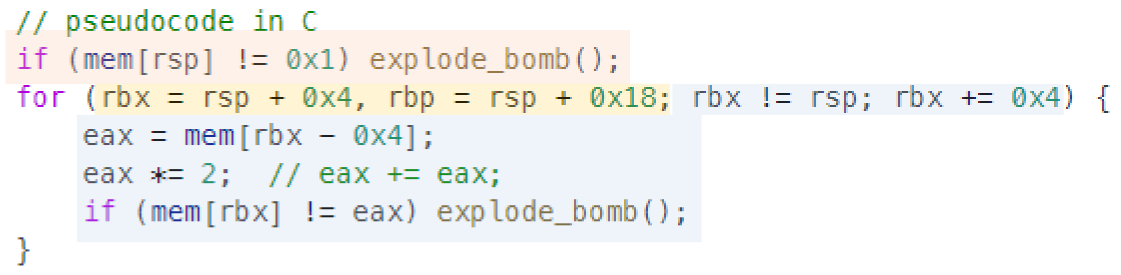
- 调用read_six_numbers，读取6个整数
  - 最先判定的整数位于栈顶（注意栈是向下增长的）
  - 这个数**必须是0x1**，否则会触发炸弹
  - 跳转到400f30去初始化，然后跳转到400f17
  - 400f17一路执行，直到400f29进行判断
    视情况回到400f17执行，或跳转至400f3c收尾
  这是一个循环
- 大致能确定出循环体的内容：遍历2次幂



```
0000000000400efc <phase_2>:
  400efc: 55                      push   %rbp
  400efd: 53                      push   %rbx
  400efe: 48 83 ec 28             sub    $0x28,%rsp
  400f02: 48 89 e6                mov    %rsp,%rsi
  400f05: e8 52 05 00 00          callq  40145c <read_six_numbers>
  400f0a: 83 3c 24 01             cmpl   $0x1,(%rsp)
  400f0e: 74 20                   je     400f30 <phase_2+0x34>
  400f10: e8 25 05 00 00          callq  40143a <explode_bomb>
  400f15: eb 19                   jmp    400f30 <phase_2+0x34>
  400f17: 8b 43 fc                mov    -0x4(%rbx),%eax
  400f1a: 01 c0                   add    %eax,%eax
  400f1c: 39 03                   cmp    %eax,(%rbx)
  400f1e: 74 05                   je     400f25 <phase_2+0x29>
  400f20: e8 15 05 00 00          callq  40143a <explode_bomb>
  400f25: 48 83 c3 04             add    $0x4,%rbx
  400f29: 48 39 eb                cmp    %rbp,%rbx
  400f2c: 75 e9                   jne    400f17 <phase_2+0x1b>
  400f2e: eb 0c                   jmp    400f3c <phase_2+0x40>
  400f30: 48 8d 5c 24 04          lea    0x4(%rsp),%rbx
  400f35: 48 8d 6c 24 18          lea    0x18(%rsp),%rbp
  400f3a: eb db                   jmp    400f17 <phase_2+0x1b>
  400f3c: 48 83 c4 28             add    $0x28,%rsp
  400f40: 5b                      pop    %rbx
  400f41: 5d                      pop    %rbp
  400f42: c3                      retq
```

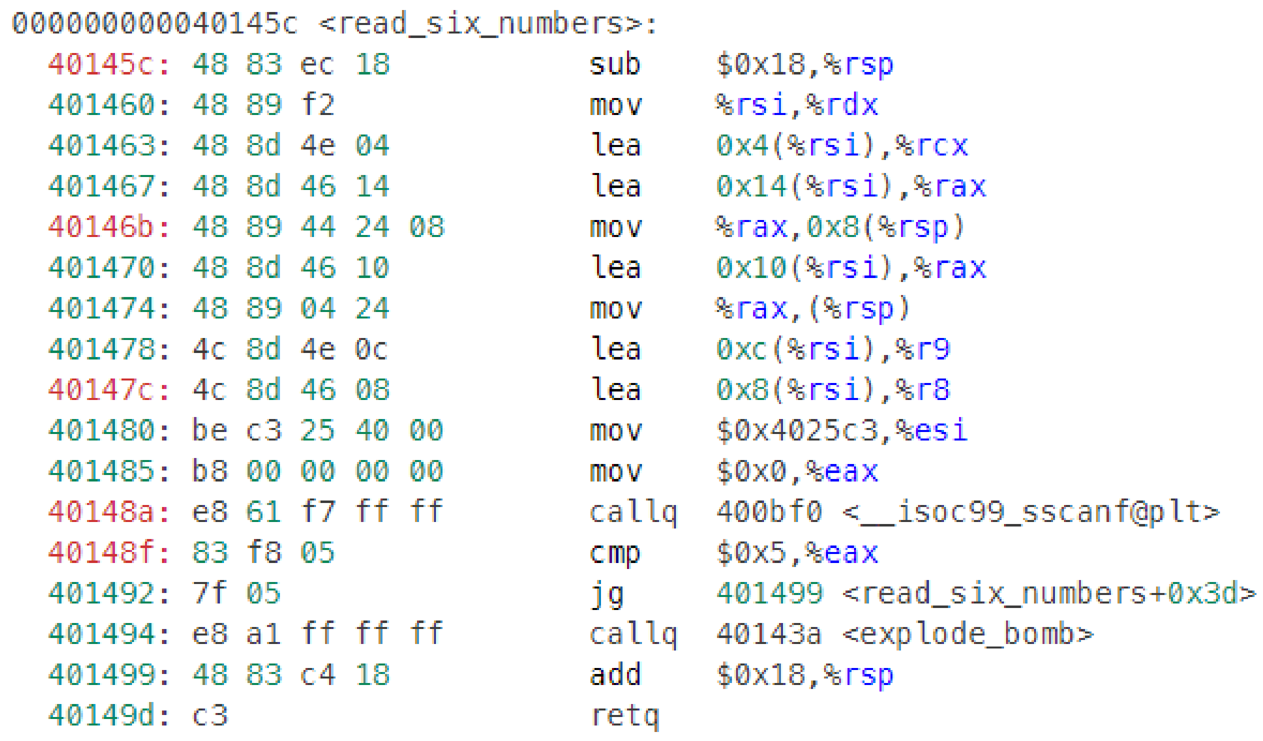


```python
# pseudocode in Python
if mem[rsp] != 0x1:
    explode_bomb()
rbx = rsp + 0x4
rbp = rsp + 0x18
while rbx != rbp:
    eax = mem[rbx - 0x4]
    eax *= 2  # eax += eax
    if mem[rbx] != eax:
        explode_bomb()
    rbx += 0x4
```



```c
// pseudocode in C
if (mem[rsp] != 0x1) explode_bomb();
for (rbx = rsp + 0x4, rbp = rsp + 0x18; rbx != rsp; rbx += 0x4) {
    eax = mem[rbx - 0x4];
    eax *= 2;  // eax += eax;
    if (mem[rbx] != eax) explode_bomb();
}
```

Bomb Lab · Phase 2

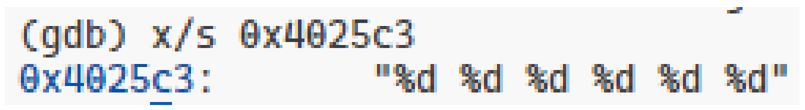


```
0000000000040145c <read_six_numbers>:
  40145c: 48 83 ec 18             sub    $0x18,%rsp
  401460: 48 89 f2                mov    %rsi,%rdx
  401463: 48 8d 4e 04             lea    0x4(%rsi),%rcx
  401467: 48 8d 46 14             lea    0x14(%rsi),%rax
  40146b: 48 89 44 24 08          mov    %rax,0x8(%rsp)
  401470: 48 8d 46 10             lea    0x10(%rsi),%rax
  401474: 48 89 04 24             mov    %rax,(%rsp)
  401478: 4c 8d 4e 0c             lea    0xc(%rsi),%r9
  40147c: 4c 8d 46 08             lea    0x8(%rsi),%r8
  401480: be c3 25 40 00          mov    $0x4025c3,%esi
  401485: b8 00 00 00 00          mov    $0x0,%eax
  40148a: e8 61 f7 ff ff          callq  400bf0 <__isoc99_sscanf@plt>
  40148f: 83 f8 05                cmp    $0x5,%eax
  401492: 7f 05                   jg     401499 <read_six_numbers+0x3d>
  401494: e8 a1 ff ff ff          callq  40143a <explode_bomb>
  401499: 48 83 c4 18             add    $0x18,%rsp
  40149d: c3                      retq
```

- 问题是应该构建怎样的输入串呢？
  - 进入read_six_numbers
    可以看到是通过sscanf()来读入的
  - sscanf读取后将结果保存在栈上
    向前兼容X86-32，变量地址自右向左入栈
    顺序输入就可以了

- int sscanf(const char* s, const char* format, ...)
- sscanf调用前，%rsi被赋值为$0x4025c3
  而%rsi保存的就是格式串的地址

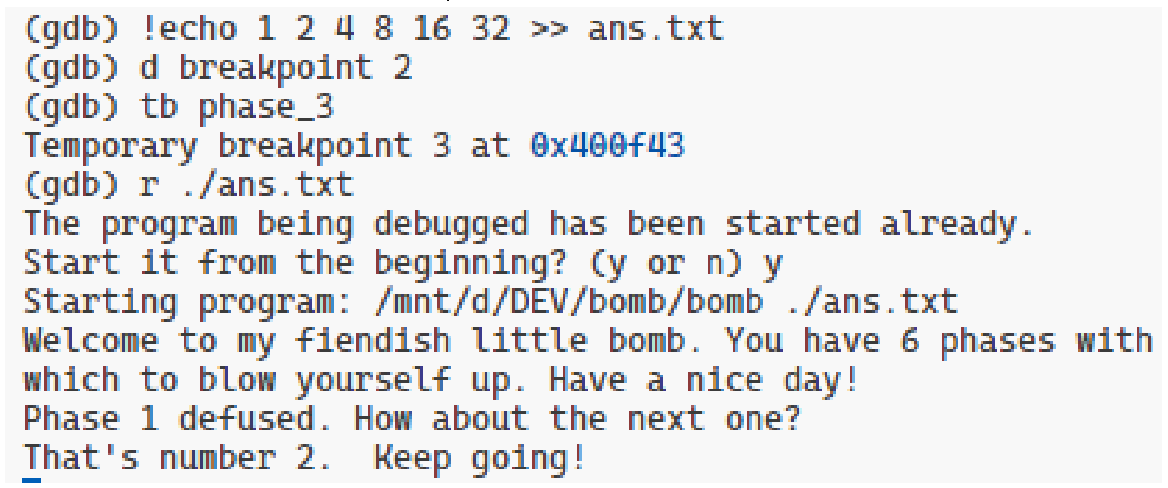
查看$0x4025c3的内容得知输入串的格式



```
(gdb) x/s 0x4025c3
0x4025c3:        "%d %d %d %d %d %d"
```

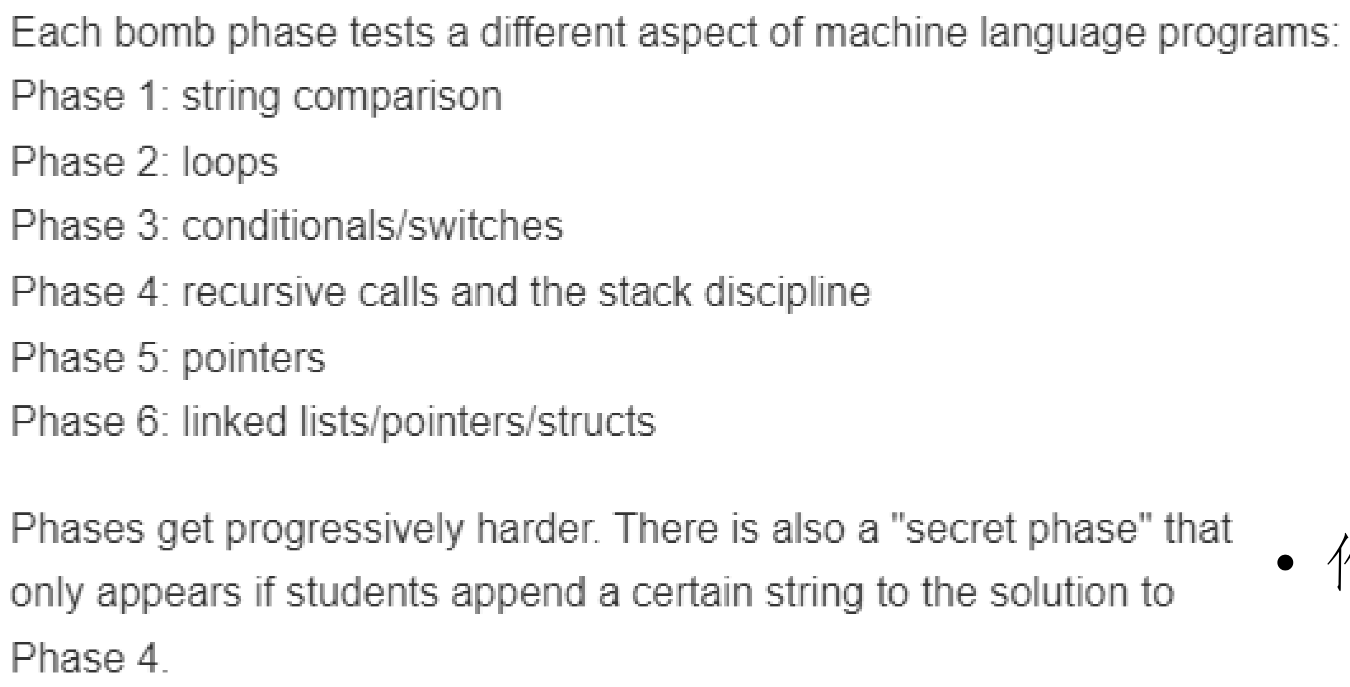
- 空格分隔的6个整数

构造 **1 2 4 8 16 32** 字符串



```
(gdb) !echo 1 2 4 8 16 32 >> ans.txt
(gdb) d breakpoint 2
(gdb) tb phase_3
Temporary breakpoint 3 at 0x400f43
(gdb) r ./ans.txt
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /mnt/d/DEV/bomb/bomb ./ans.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
```

Bomb Lab · 后续Phase



Each bomb phase tests a different aspect of machine language programs:

Phase 1: string comparison

Phase 2: loops

Phase 3: conditionals/switches

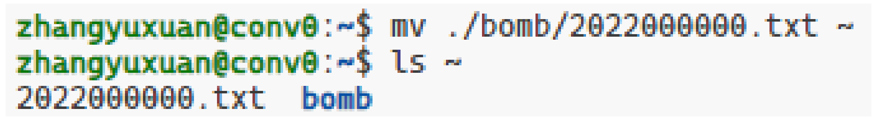
Phase 4: recursive calls and the stack discipline

Phase 5: pointers

Phase 6: linked lists/pointers/structs

Phases get progressively harder. There is also a "secret phase" that only appears if students append a certain string to the solution to Phase 4.

- 作业提交：
  - 每一关卡的“通关答案”，保存在 **学号.txt**中
    最后移动到家目录~ 下

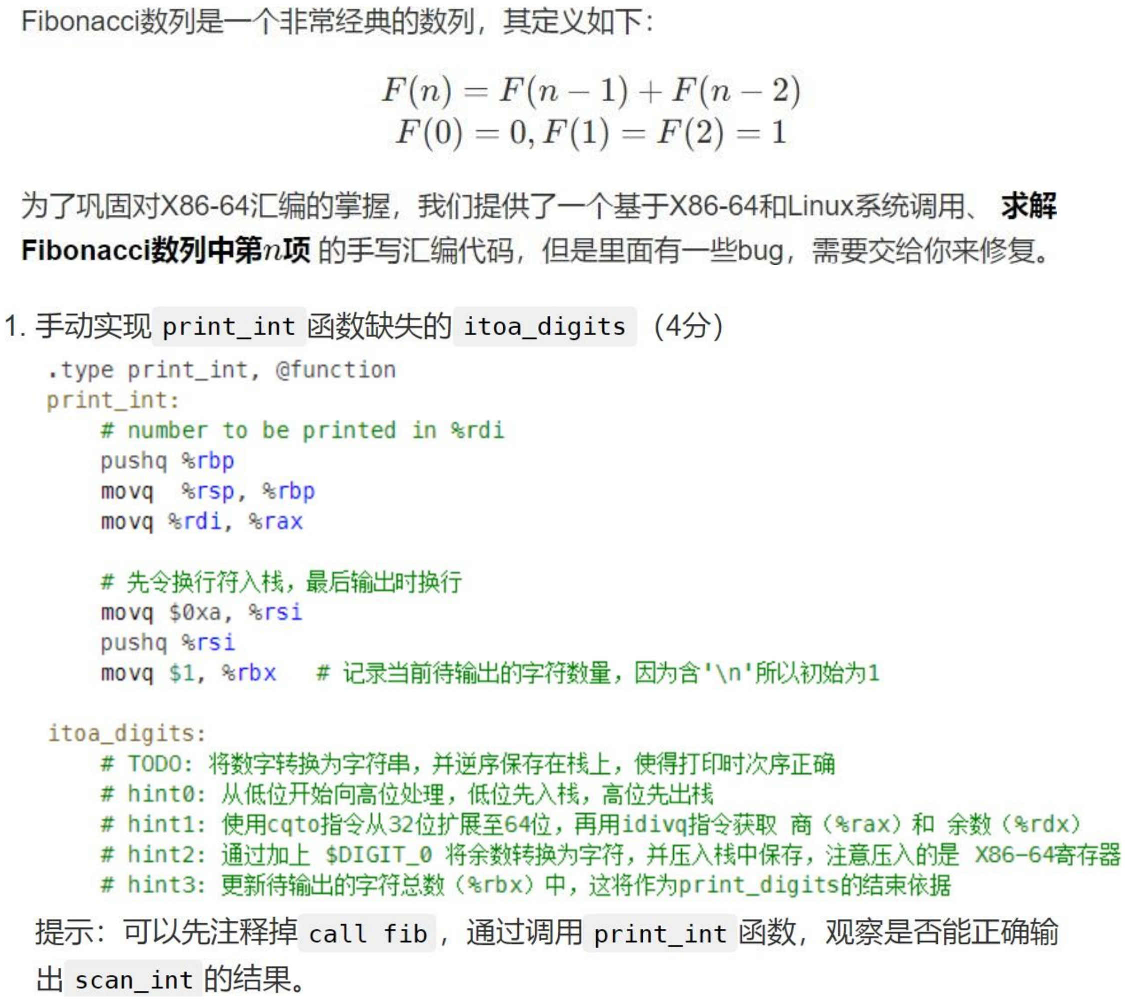


```
zhangyuxuan@conv0:~$ mv ./bomb/2022000000.txt ~
zhangyuxuan@conv0:~$ ls ~
2022000000.txt  bomb
```

  - 一份不多于**8**页的题解(writeup)，简述你破解每个phase的方式，提交到网络学堂
    鼓励大家使用Markdown来编写
  - **不查重 (!)，每个人都是独一无二的（笑）**
  - **DDL 为 11月23日（星期四）23:59**
    迟交1天，总分扣10%，直至0分
- 作业评分：
  - 学号.txt 60%：一个 phase 10%
  - writeup 40%：
    每个phase的解题思路和结果　6 * 5%
    感想/吐槽/新的实验设计　　10%

Bomb Lab · 难度太大？

- 可以先试试难度较低的实践题



Fibonacci数列是一个非常经典的数列，其定义如下：

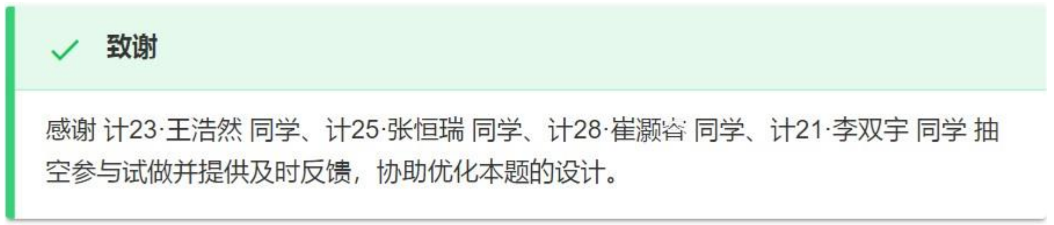
$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = 0, F(1) = F(2) = 1$$

为了巩固对X86-64汇编的掌握，我们提供了一个基于X86-64和Linux系统调用、 **求解Fibonacci数列中第n项** 的手写汇编代码，但是里面有一些bug，需要交给你来修复。

1. 手动实现 `print_int` 函数缺失的 `itoa_digits` （4分）

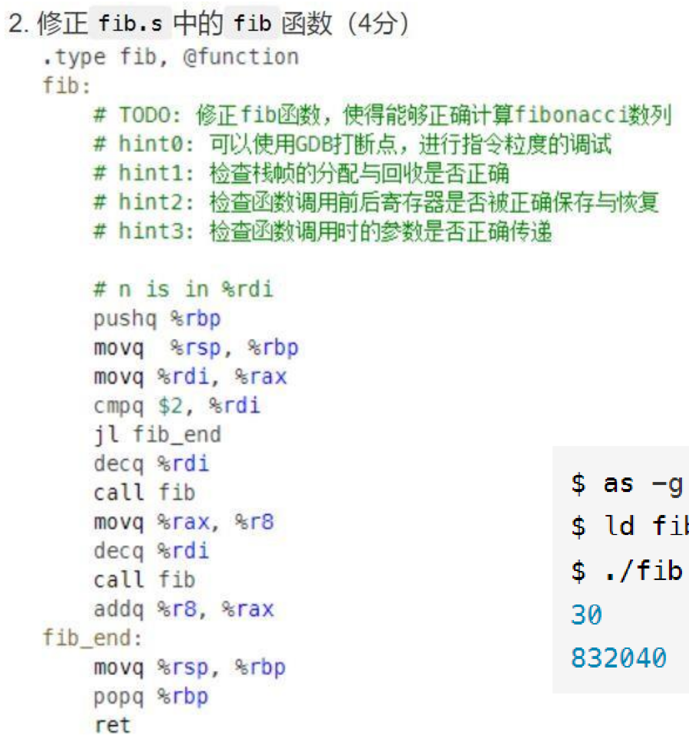
```
.type print_int, @function
print_int:
    # number to be printed in %rdi
    pushq %rbp
    movq  %rsp, %rbp
    movq %rdi, %rax

    # 先令换行符入栈，最后输出时换行
    movq $0xa, %rsi
    pushq %rsi
    movq $1, %rbx    # 记录当前待输出的字符数量，因为含'\n'所以初始为1

itoa_digits:
    # TODO: 将数字转换为字符串，并逆序保存在栈上，使得打印时次序正确
    # hint0: 从低位开始向高位处理，低位先入栈，高位先出栈
    # hint1: 使用cqto指令从32位扩展至64位，再用idivq指令获取 商（%rax）和 余数（%rdx）
    # hint2: 通过加上 $DIGIT_0 将余数转换为字符，并压入栈中保存，注意压入的是 X86-64寄存器
    # hint3: 更新待输出的字符总数（%rbx）中，这将作为print_digits的结束依据
```

提示：可以先注释掉 `call fib` ，通过调用 `print_int` 函数，观察是否能正确输出 `scan_int` 的结果。



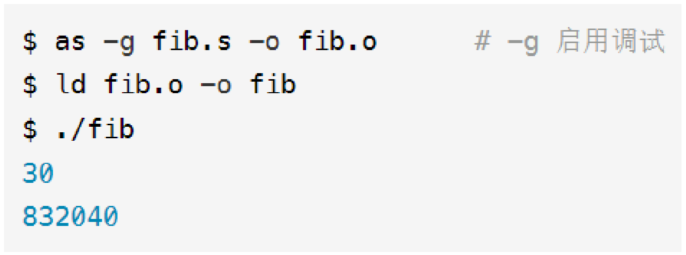
✓ **致谢**

感谢 计23·王浩然 同学、计25·张恒瑞 同学、计28·崔灏容 同学、计21·李双宇 同学 抽空参与试做并提供及时反馈，协助优化本题的设计。



2. 修正 `fib.s` 中的 `fib` 函数（4分）

```
.type fib, @function
fib:
    # TODO: 修正fib函数，使得能够正确计算fibonacci数列
    # hint0: 可以使用GDB打断点，进行指令粒度的调试
    # hint1: 检查栈帧的分配与回收是否正确
    # hint2: 检查函数调用前后寄存器是否被正确保存与恢复
    # hint3: 检查函数调用时的参数是否正确传递

    # n is in %rdi
    pushq %rbp
    movq  %rsp, %rbp
    movq %rdi, %rax
    cmpq $2, %rdi
    jl fib_end
    decq %rdi
    call fib
    movq %rax, %r8
    decq %rdi
    call fib
    addq %r8, %rax
fib_end:
    movq %rsp, %rbp
    popq %rbp
    ret
```



```
$ as -g fib.s -o fib.o      # -g 启用调试
$ ld fib.o -o fib
$ ./fib
30
832040
```

计算机系统概论

GDB 进阶用法

陈嘉杰　张学峰　张宇轩

清华大学《计算机系统概论》课教学团队

2023

一些约定

一些约定：

1. 形如 a{,b,c,d}e 的表示方法，是 ae,abe,ace,ade 的一个缩写。
2. 涉及到执行命令的代码框，如果行首有 $，那么后面表示的是执行的命令；否则，表示的是命令的输出。

实验环境准备

1. 需要是 x86_64 架构的 Linux 系统
2. 如果是 Windows 建议安装 WSL2，在 WSL 内进行实验
3. 如果是 macOS，请 SSH 到 Linux 机器上进行实验
4. 首先安装 gcc 和 gdb 和 objdump，以 Debian 为例子：`sudo apt install gcc gdb binutils`
5. 检验 gdb 是否安装了：`which gdb` 应该会输出 gdb 的路径

C/C++ 代码编译流程

一段 C/C++ 代码编译成二进制的可执行文件，需要下面四个步骤：

1. 预处理（Preprocess）：处理 #include 和宏
2. 编译（Compile）：将 C 代码编译成汇编代码
3. 汇编（Assemble）：将汇编代码编译成指令，保存在对象文件中
4. 链接（Link）：将对象文件和标准库链接为二进制可执行文件

实际编译的时候，调用编译器时，根据命令行参数的不同，会执行上面的部分或者所有步骤。
常用的 C/C++ 编译器有：GCC、Clang、MSVC、ICC。本讲以 GCC 为例子。

GCC 使用

如果要在 GCC 里面编译一个程序，最简单的办法就是 `gcc a.c -o a`，但也可以一步一步地进行四个步骤的生成：

1. 预处理：`gcc -E a.c -o a.i`
2. 编译：`gcc -S a.i -o a.s`
3. 汇编：`gcc -c a.s -o a.o`
4. 链接：`gcc a.o -o a`

举个例子

比如，我们编写一个简单的程序如下，保存为 a.c：

```c
#define RET 0
int main(int argc, char *argv[]) {
  return RET;
}
```

举个例子

首先进行预处理 `gcc -E a.c -o a.i`，可以得到这样的内容：

```
# 1 "a.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "a.c"

int main(int argc, char *argv[]) {
 return 0;
}
```

可以看到代码中的宏已经展开。如果 #include 了头文件，头文件的内容也会出现在这里，同学们可以自己进行实验。

举个例子

接着，编译代码到汇编 `gcc -S a.i -o a.s`，可以得到这样的内容，省略了一些注释：

```
1          .text
2          .globl  main
3  main:
4          pushq   %rbp
5          movq    %rsp, %rbp
6          movl    %edi, -4(%rbp)
7          movq    %rsi, -16(%rbp)
8          movl    $0, %eax
9          popq    %rbp
10         ret
```

我们编写的代码编译成了 AT&T 风格的 x86_64 汇编；可以看到，第 4-7 行维护了栈指针，并且把前两个参数（%edi %rsi）保存在栈上，第 9-10 行设置返回值（%eax）为 0，然后返回。

举个例子

得到汇编以后，需要汇编为二进制的指令，并保存在对象文件中：`gcc -c a.s -o a.o`。可以用 `file a.o` 来查看 a.o 文件的格式，得到：

```
$ gcc -c a.s -o a.o
$ file a.o
a.o: ELF 64-bit LSB relocatable, x86-64, version 1
↪   (SYSV), not stripped
```

接着，链接为可执行程序：

```
$ gcc a.o -o a
$ file a
a: ELF 64-bit LSB pie executable, x86-64, version 1
↪   (SYSV), dynamically linked, interpreter
↪   /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
↪   not stripped
```

小贴士：Windows 下可执行文件后缀一般是 exe/msi，其他平台下可执行文件没有后缀。

GCC 总结

总结一下 GCC 的一些参数用法，也给出了其他一些常用的参数：

-o filename	指定输出的文件名
-S	输出汇编代码
-c	编译到对象文件（object file，.o/.obj）
-E	对代码进行预处理
-g	打开调试符号
-Werror	把所有的 warning 当成 error
-O{,1,2,3,s,fast,g}	设置优化选项

为了调试方便，一般需要打开 -g 选项，否则调试的时候可能会遇到无法设置断点，或者在断点无法看到变量信息等问题。

objdump 使用

我们需要使用 objdump 工具来查看编译出来的汇编，例子如下：

```
$ objdump -S a.o
a.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d fc                mov    %edi,-0x4(%rbp)
   7:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
   b:   b8 00 00 00 00          mov    $0x0,%eax
  10:   5d                      pop    %rbp
  11:   c3                      retq
```

可以看到，我们定义的 main 函数被编译成了上面的 7 条指令，这个内容与我们之前看到的 a.s 基本是一致的。区别在于，每条指令出现了地址、二进制表示，并且指令的表示也有细微的差别。

objdump 使用

刚刚只用 objdump 查看了 a.o 对应的汇编，我们再看看 a 可执行文件的汇编：

```
$ objdump -S a
a:      file format elf64-x86-64
Disassembly of section .init:
0000000000001000 <_init>:
  1000:  48 83 ec 08             sub    $0x8,%rsp
  1004:  48 8b 05 dd 2f 00 00    mov
  ↪  0x2fdd(%rip),%rax  # 3fe8 <__gmon_start__>
  100b:  48 85 c0                test   %rax,%rax
...
```

因为输出比较长，省略了一些部分输出。可以看到，这里出现了很多没有见过的代码，这些代码来自于 libc，在链接的时候，a.o 和 libc 的一些代码共同组成了 a 的二进制代码。

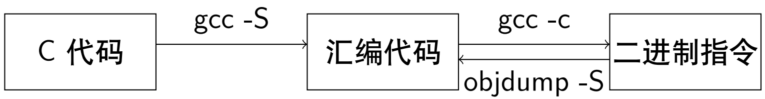
objdump 使用

如果我们继续往下找 main 函数：

```
0000000000001125 <main>:
  1125:   55                      push   %rbp
  1126:   48 89 e5                mov    %rsp,%rbp
...
  1135:   5d                      pop    %rbp
  1136:   c3                      retq
  1137:   66 0f 1f 84 00 00 00    nopw
    ↪  0x0(%rax,%rax,1)
  113e:   00 00
```

可以发现 main 函数的地址变了，因为前面出现了其他函数；另外，可以看到 return 之后还有 nop 指令，这是为了让下一个函数的起始地址对齐到某个数（比如 16）的整数倍上。

objdump 与 GCC 关系



objdump 常用参数

-d	反汇编可执行的段（如.text）
-D	反汇编所有的段，即使这个段存储的是数据，也当成指令来解析
-S	把反汇编的代码和调试信息里的代码信息合在一起显示
-t	显示函数的符号列表
--adjust-vma OFFSET	把反汇编出来的地址都加上一个偏移，常用于嵌入式开发
-M {intel,att}	用 Intel 或者 AT&T 风格显示 X86 汇编；默认情况下用的是 AT&T 风格

如果用的是 MSVC，那么默认的 X86 汇编风格是 Intel 风格；如果用的是 objdump、gdb、gcc 等工具，一般默认的汇编风格是 AT&T。

gdb 介绍

gdb 是一个调试器，可以在运行程序的时候，设置断点，中途查看程序运行的状态，并且逐步观察程序运行行为。
我们引入下面的代码作为例子，来观察 gdb 的使用方式：

```c
#include <stdio.h>
int number;
int main(int argc, char *argv[]) {
  printf("%d %s\n", argc, argv[0]);
  scanf("%d", &number);
  printf("%d\n", number);
  return 0;
}
```

按照前面讲述的方法，编译成二进制：`g++ test.cpp -o test`

gdb 介绍

用 gdb 调试 test 程序，首先运行 gdb test，然后输入 run 命令开始运行：

```
$ gdb test
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
...
Reading symbols from test...
(No debugging symbols found in test)
(gdb) run
Starting program: /home/jiegec/test
1 /home/jiegec/test
1234
1234
[Inferior 1 (process 1773151) exited normally]
(gdb)
```

程序运行以后，输出了 argc argv，这时候可以正常向程序输入数据。程序退出以后，回到 gdb 的命令。接下来，我们要设置断点，来观察程序的行为。

gdb 介绍

首先，我们可以用 b（全称 breakpoint）来设置一个断点，比如这里给 main 函数设置一个断点。

```
(gdb) b main
Breakpoint 1 at 0x555555555149
(gdb) run
Starting program: /home/jiegec/test
Breakpoint 1, 0x0000555555555149 in main ()
(gdb) c
Continuing.
1 /home/jiegec/test
1234
1234
[Inferior 1 (process 1775170) exited normally]
(gdb)
```

可以看到，程序在 main 函数的开头暂停了运行，回到了 gdb，并且可以看到当前运行的指令地址和函数。但由于我们在编译的时候没有打开调试信息，输入 c（全称 continue）命令让它继续运行到结束。

gdb 介绍

那么，我们打开调试选项重新编译一次代码 g++ -g test.cpp -o test，可以发现 gdb 找到了调试符号：

```
$ gdb test
Reading symbols from test...
(gdb)
```

这时候再设置断点，并且运行，就可以看到程序停在了断点所在的地方。

```
(gdb) b main
Breakpoint 1 at 0x1154: file test.cpp, line 4.
(gdb) run
Starting program: /home/jiegec/test
Breakpoint 1, main (argc=1, argv=0x7fffffffeaf8) at
 ↪  test.cpp:4
4               printf("%d %s\n", argc, argv[0]);
(gdb)
```

gdb 介绍

我们可以用 n 命令来执行到下一行代码，也可以用 p 命令来打印出变量的值：

```
(gdb) n
1 /home/jiegec/test
5                 scanf("%d", &number);
(gdb) p argc
$1 = 1
(gdb) p argv
$2 = (char **) 0x7fffffffeaf8
(gdb) p argv[0]
$3 = 0x7fffffffed3d "/home/jiegec/test"
(gdb) p argv[1]
$4 = 0x0
```

gdb 介绍

继续运行，scanf 结束后，可以看到 number 的值和我们输入是一致的：

```
(gdb) n
12345
6                   printf("%d\n", number);
(gdb) p number
$5 = 12345
(gdb) n
12345
7                   return 0;
(gdb)
```

gdb 介绍

上面几个基础的命令已经可以实现很多程序的调试，下面再总结一下常用的 gdb 命令：

run	运行程序，后面可以跟上运行程序的命令行参数
b func/file:line/*addr	设置断点
s	如果当前行是函数调用，进入函数调用；否则执行当前行代码，进入下一行
n	执行当前行代码，进入下一行
si,ni	和上面两个命令的不同是，它的粒度是指令
q	退出程序
p var/$reg	输出代码中的变量或者寄存器
info registers	输出所有寄存器的值
disas func/addr/$reg	输出目标函数/地址的汇编
disas /m func/addr/$reg	输出目标函数/地址的汇编和源代码
x/5i func/addr/$reg	输出目标函数/地址的前 5 条汇编
layout src	显示源代码窗口
layout asm	显示汇编窗口