

C 语言拾遗(1): 机制

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



讲前提醒

PA0即将截止:

- * 2022年9月18日23:59:59 (以此 deadline 计按时提交bonus)

PA1已悄悄发布:

- * PA 1.1: 2022.9.25 (此为建议的不计分 deadline)

- * PA 1.2: 2022.10.2 (此为建议的不计分 deadline)

- * PA 1.3: 2022.10.9 23:59:59 (以此 deadline 计按时提交bonus)

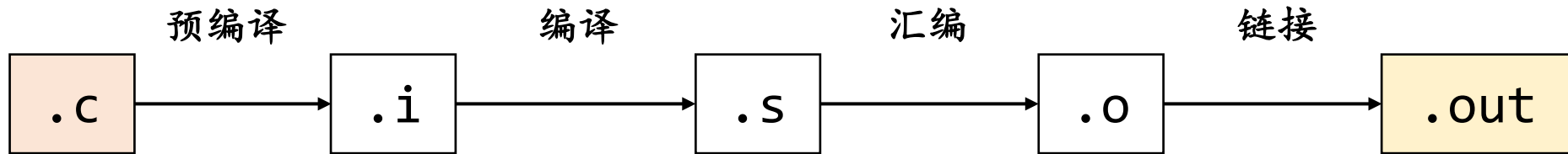
本讲概述

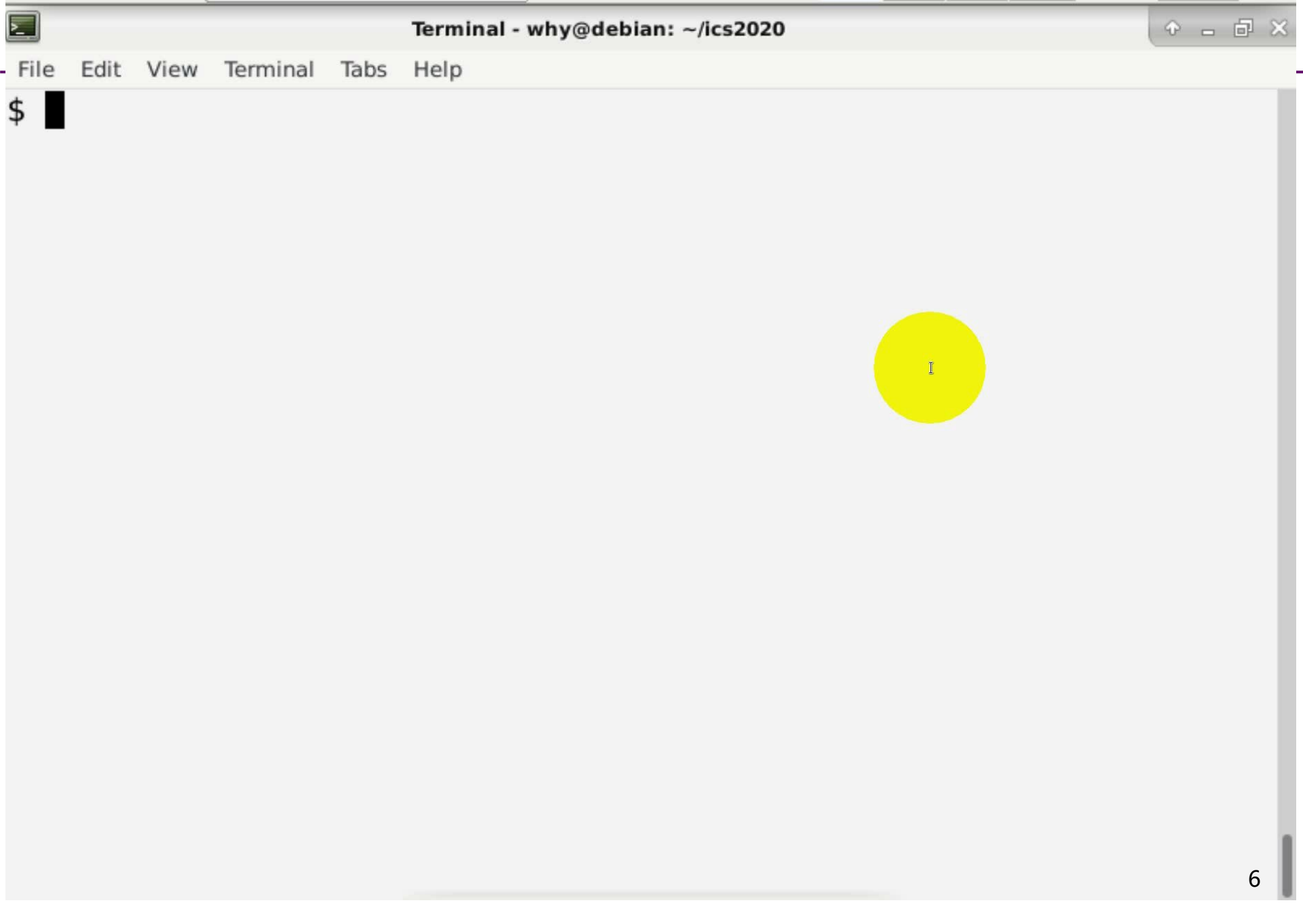
- 在IDE里，为什么按一个键，就能够编译运行？
 - 编译、链接
 - `.c` → 预编译 → `.i` → 编译 → `.s` → 汇编 → `.o` → 链接 → `a.out`
 - 加载执行
 - `./a.out`
- 背后是通过调用命令行工具完成的
 - RTFM: `man gcc`; `gcc -help`; `tlldr gcc`
 - 控制行为的三个选项: `-E`, `-S`, `-c`
- 本次课程
 - 预热: 编译、链接、加载到底做了什么?
 - RTFSC时需要关注的C语言特性

IDE的一个键到底发生了什么？

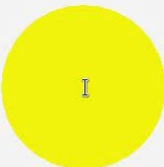


IDE的一个键到底发生了什么？

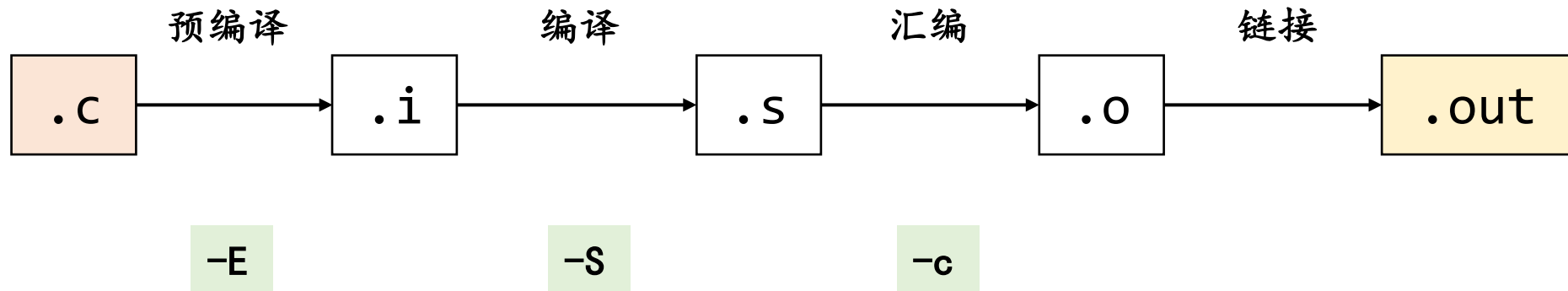




\$



从源代码到可执行文件



进入C语言之前：预编译

```
#include  
#define  
#  
##  
#ifdef
```

#include指令

- 什么是#include?
- 怎么理解?



#include<>指令

- 以下代码有什么区别？

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

- 为什么在没有安装库时会发生错误？

```
#include <SDL/SDL2.h>
```

- 你可能在书/阅读材料上了解过一些相关的知识
 - 但更好的办法是阅读命令的日志
 - gcc --verbose a.c

```
$ ls
a.c a.inc a.out a.s b
$
```

#include<>指令

- 以下代码有什么区别？

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

- 为什么在没有安装库时会发生错误？

```
#include <SDL/SDL2.h>
```

- 你可能在书/阅读材料上了解过一些相关的知识
 - 但更好的办法是[阅读命令的日志](#)
 - gcc --verbose a.c

```
#include "..." search starts here:
#include <...> search starts here:
 /usr/lib/gcc/x86_64-linux-gnu/8/include
 /usr/local/include
 /usr/lib/gcc/x86_64-linux-gnu/8/include-fixed
 /usr/include/x86_64-linux-gnu
 /usr/include
```

\$

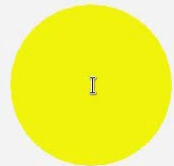
有趣的预编译

- 以下代码会输出什么?
 - 为什么?

```
#include <stdio.h>

int main() {
    #if aa == bb
        printf("Yes\n");
    #else
        printf("No\n");
    #endif
}
```

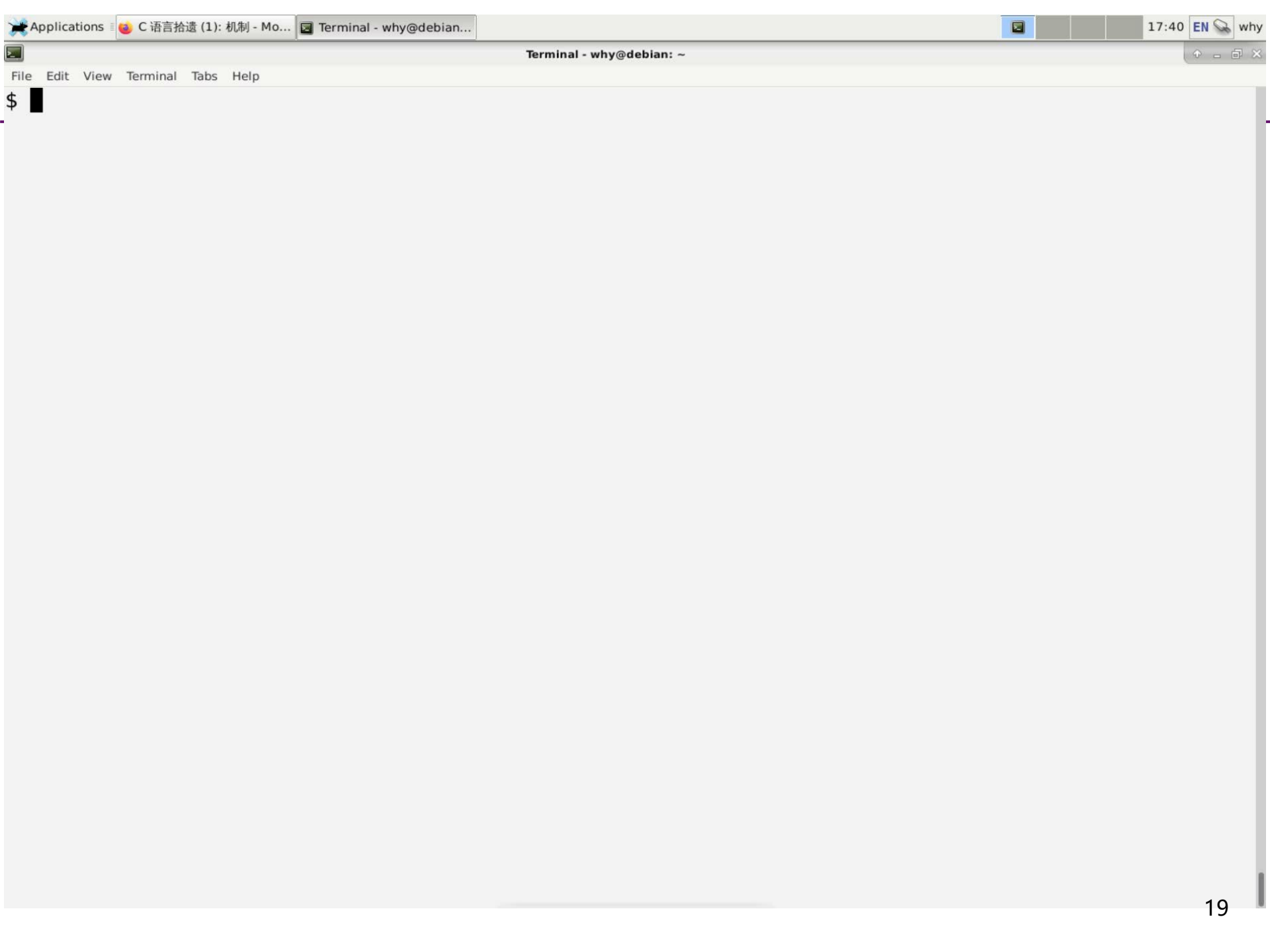

\$



宏定义与展开

- 宏展开：通过复制/粘贴改变代码的形态
 - #include → 粘贴文件
 - aa, bb → 粘贴符号
- 知乎问题：如何搞垮一个OJ?

```
#define A "aaaaaaaaaa"
#define TEN(A) A A A A A A A A A A
#define B TEN(A)
#define C TEN(B)
#define D TEN(C)
#define E TEN(D)
#define F TEN(E)
#define G TEN(F)
int main() { puts(G);}
```



宏定义与展开

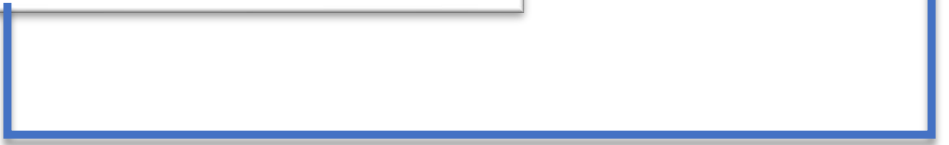
- 如何躲过Online Judge的关键字过滤？

```
#define SYSTEM sys ## tem
```

File Edit View Terminal Tabs Help

```
1 #define A sys ## tem
2
3 int main(){
4     A("echo Hello\n");
5 }
```

```
$ ./a.out
Hello
```

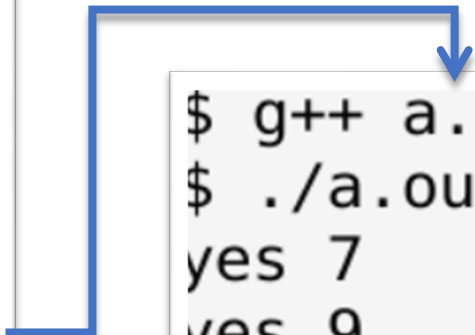


宏定义与展开

- 如何毁掉一个身边的同学?

```
#define true (__LINE__ % 2 != 0)
```

```
File Edit View Terminal Tabs Help
1 #define true (__LINE__ % 2 != 0)
2
3 #include <stdio>
4
5 int main(){
6     if (true) printf("yes %d\n", __LINE__);
7     if (true) printf("yes %d\n", __LINE__);
8     if (true) printf("yes %d\n", __LINE__);
9     if (true) printf("yes %d\n", __LINE__);
10    if (true) printf("yes %d\n", __LINE__);
11    if (true) printf("yes %d\n", __LINE__);
12    if (true) printf("yes %d\n", __LINE__);
13    if (true) printf("yes %d\n", __LINE__);
14    if (true) printf("yes %d\n", __LINE__);
15    if (true) printf("yes %d\n", __LINE__);
16    if (true) printf("yes %d\n", __LINE__);
17    if (true) printf("yes %d\n", __LINE__);
18    if (true) printf("yes %d\n", __LINE__);
19    if (true) printf("yes %d\n", __LINE__);
20    if (true) printf("yes %d\n", __LINE__);
21 }
```



```
$ g++ a.c
$ ./a.out
yes 7
yes 9
yes 11
yes 13
yes 15
yes 17
yes 19
```

宏定义与展开

```
#define s ((((((((((((((( 0
#define _ * 2)
#define X * 2 + 1)
static unsigned short stopwatch[] = {
    s _ _ _ _ _ X X X X X _ _ _ X X _ ,
    s _ _ _ X X X X X X X X X _ X X X ,
    s _ _ X X X _ _ _ _ _ X X X _ X X ,
    s _ X X _ _ _ _ _ _ _ _ X X _ _ ,
    s X X _ _ _ _ _ _ _ _ _ _ X X _ ,
    s X X _ X X X X X _ _ _ _ _ X X _ ,
    s X X _ _ _ _ _ X _ _ _ _ _ X X _ ,
    s X X _ _ _ _ _ X _ _ _ _ _ X X _ ,
    s _ X X _ _ _ _ X _ _ _ _ X X _ _ ,
    s _ _ X X X _ _ _ _ _ X X X _ _ _ ,
    s _ _ _ X X X X X X X X X _ _ _ _ ,
    s _ _ _ _ _ X X X X X _ _ _ _ _ , };
```

宏定义与展开

```
gcc -E a.c
```

```
#define s (((((((((((((((((( 0
#define _ * 2)
#define X * 2 + 1)
static unsigned short stopwatch[] = {
s _ _ _ _ _ X X X X X _ _ _ X X _ ,
s _ _ _ _ _ X X X X X X X _ _ _ X X X ,
s _ _ _ X X X _ _ _ _ _ X X X _ X X ,
s _ _ _ X X _ _ _ _ _ _ _ _ _ X X _ ,
s X X _ _ _ _ _ _ _ _ _ _ _ X X _ ,
s X X _ _ X X X X X _ _ _ _ _ X X _ ,
s X X _ _ _ _ _ X _ _ _ _ _ X X _ ,
s X X _ _ _ _ _ X _ _ _ _ _ X X _ ,
s _ _ X X _ _ _ _ _ X _ _ _ _ _ X X _ ,
s _ _ _ X X X _ _ _ _ _ X X X _ _ _ ,
s _ _ _ _ X X X X X X X X X _ _ _ ,
s _ _ _ _ _ X X X X X , };
```

```
static unsigned short stopwatch[] = {  
    (((((((((((((((((( 0 * 2) * 2) * 2) * 2) * 2) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1) * 2) * 2  
    ) * 2) * 2 + 1) * 2 + 1) * 2),  
    (((((((((((((((((( 0 * 2) * 2) * 2) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1)  
    * 2 + 1) * 2 + 1) * 2) * 2 + 1) * 2 + 1) * 2 + 1),  
    (((((((((((((((((( 0 * 2) * 2) * 2) * 2 + 1) * 2 + 1) * 2 + 1) * 2) * 2) * 2) * 2) * 2) * 2 + 1) * 2 + 1  
    ) * 2 + 1) * 2) * 2 + 1) * 2 + 1),  
    (((((((((((((((((( 0 * 2) * 2 + 1) * 2 + 1) * 2) * 2) * 2) * 2) * 2) * 2) * 2) * 2) * 2) * 2) * 2 + 1) *  
    2 + 1) * 2) * 2),  
    (((((((((((((((((( 0 * 2 + 1) * 2 + 1) * 2) * 2) * 2) * 2) * 2) * 2) * 2) * 2) * 2) * 2) * 2) * 2) * 2 +  
    1) * 2 + 1) * 2),  
    (((((((((((((((((( 0 * 2 + 1) * 2 + 1) * 2) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1) * 2) * 2)  
    * 2) * 2) * 2) * 2 + 1) * 2 + 1) * 2),  
    (((((((((((((((((( 0 * 2 + 1) * 2 + 1) * 2) * 2) * 2) * 2) * 2) * 2) * 2 + 1) * 2) * 2) * 2) * 2) * 2) *  
    2 + 1) * 2 + 1) * 2),  
    (((((((((((((((((( 0 * 2 + 1) * 2 + 1) * 2) * 2) * 2) * 2) * 2) * 2) * 2 + 1) * 2) * 2) * 2) * 2) * 2) *  
    2 + 1) * 2 + 1) * 2),  
    (((((((((((((((((( 0 * 2) * 2 + 1) * 2 + 1) * 2) * 2) * 2) * 2) * 2) * 2 + 1) * 2) * 2) * 2) * 2) * 2 +  
    1) * 2 + 1) * 2) * 2),  
    (((((((((((((((((( 0 * 2) * 2) * 2 + 1) * 2 + 1) * 2 + 1) * 2) * 2) * 2) * 2) * 2) * 2) * 2 + 1) * 2 + 1  
    ) * 2 + 1) * 2) * 2) * 2),  
    (((((((((((((((((( 0 * 2) * 2) * 2) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1) * 2 + 1)  
    )
```

```
$ ./a.out
1990
8183
14395
24588
49158
57094
49414
49414
24844
14392
8176
1984
```

X-Macros

- 宏展开：通过复制/粘贴改变代码的形态
 - 反复粘贴，直到没有宏可以展开为止
- 例子：X-macro

```
#define NAMES(X) \  
    X(Tom) X(Jerry) X(Tyke) X(Spike)  
  
int main() {  
    #define PRINT(x) puts("Hello, " #x "!" );  
    NAMES(PRINT)  
}
```

```
$ ./a.out  
Hello, Tom!  
Hello, Jerry!  
Hello, Tyke!  
Hello, Spike!
```

PRINT(TOM) PRINT(Jerry) PRINT(Tyke) PRINT(Spike)

X-Macros

```
#define NAMES(X) \  
    X(Tom) X(Jerry) X(Tyke) X(Spike)
```

```
int main() {  
    #define PRINT(x) puts("Hello, " #x "!");  
    NAMES(PRINT)  
    #define PRINT2(x) puts("Goodbye, " #x "!");  
    NAMES(PRINT2)  
}
```

```
PRINT(TOM) PRINT(Jerry) PRINT(Tyke) PRINT(Spike)
```

```
PRINT2(TOM) PRINT2(Jerry) PRINT2(Tyke) PRINT2(Spike)
```

```
$ ./a.out  
Hello, Tom!  
Hello, Jerry!  
Hello, Tyke!  
Hello, Spike!  
Goodbye, Tom!  
Goodbye, Jerry!  
Goodbye, Tyke!  
Goodbye, Spike!
```

有趣的预编译

- 发生在实际编译之前
 - 也称为元编程 (meta-programming)
 - Gcc的预处理器同样可以处理汇编代码
 - C++中的模板元编程; Rust中的macros; ...
- Pros
 - 提供灵活的用法 (X-macros)
 - 接近自然语言的写法
- Cons
 - 破坏可读性IOCCC、程序分析 (补全)、.....

```
#define L (  
int main L ) { puts L "Hello, World" ); }
```

编译与链接

(先行剧透本学期的主要内容)

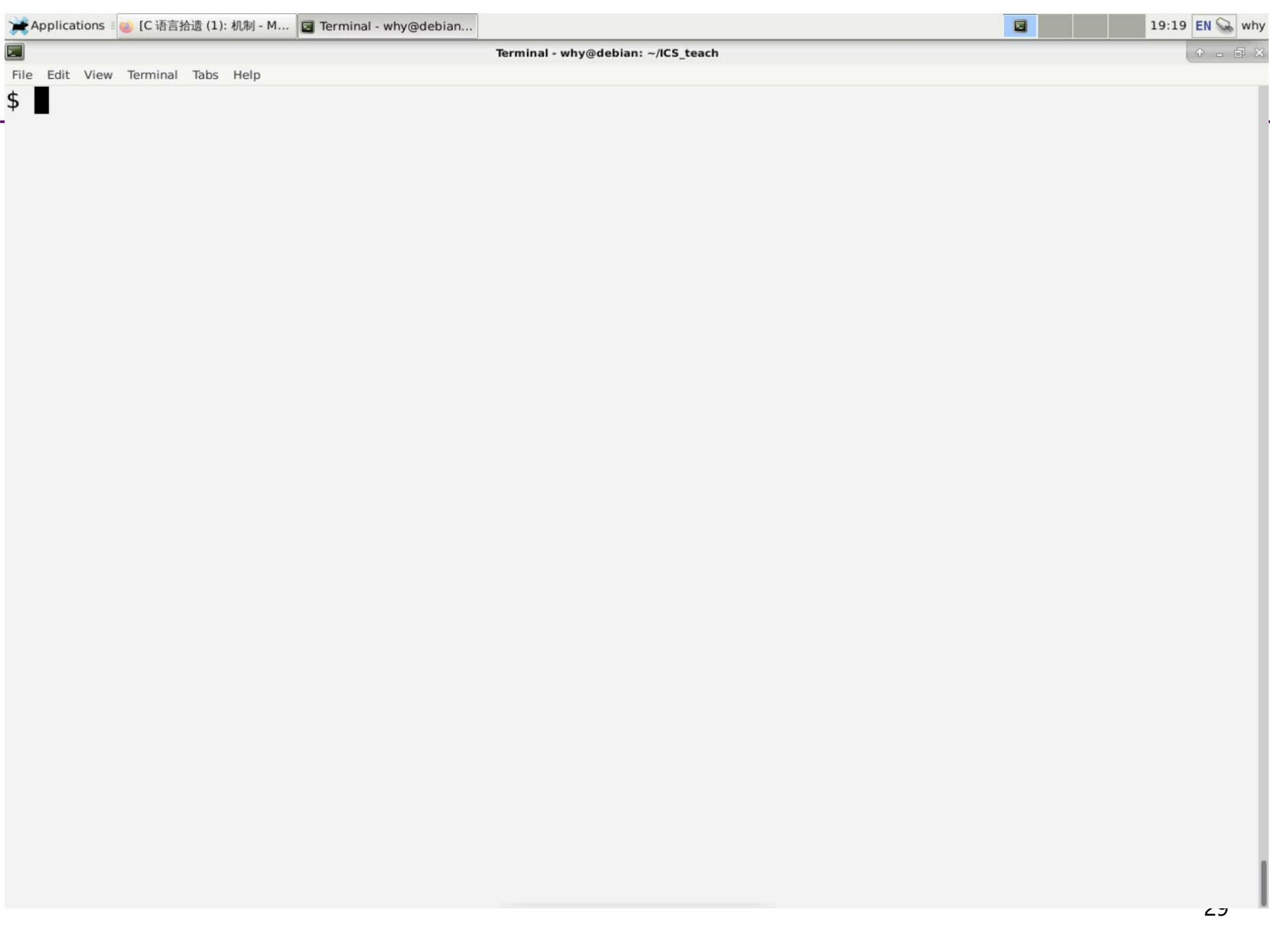
编译、链接

.c → 预编译 → .i → 编译 → .s → 汇编 → .o → 链接 → .out

编译

- 一个不带优化的简易（理想）的编译器
 - C代码中的连续一段总能找到对应的一段连续的机器指令
 - 这就是为什么大家会觉得C是高级的汇编语言！

```
int foo(int n) {  
    int sum = 0;  
    for (int i = 1; i <= n; i++) {  
        sum += i;  
    }  
    return sum;  
}
```



编译

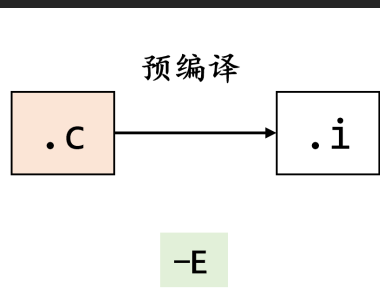
- 一个不带优化的简易（理想）的编译器
 - C代码中的连续一段总能找到对应的一段连续的机器指令
 - 这就是为什么大家会觉得C是高级的汇编语言！

```
int foo(int n) {  
    int sum = 0;  
    for (int i = 1; i <= n; i++) {  
        sum += i;  
    }  
    return sum;  
}
```

a.c到a.s到a.o

a.s

```
1 foo:
2     n = ARG-1
3     sum = 0
4     i = 1
5     goto .L2
6 .L3:
7     tmp = i
8     sum += tmp
9     i += 1
10 .L2:
11     tmp =
12     compa
13     if(<=
14     RETUR
15
16     ret
```



没有main还不能运行

a.c

```
1 int foo(int n) {
2     int sum = 0;
3     for (int i = 1; i <= n; i++) {
4         sum += i;
5     }
```

```
$ gcc -c a.c
$ objdump -d a.o
```

a.o: file format elf64-x86-64

Disassembly of section .text:

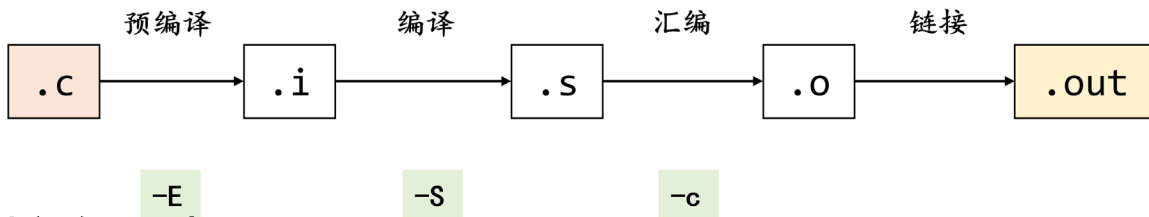
0000000000000000 <foo>:

```
0: 55
1: 48 89 e5
4: 89 7d ec
7: c7 45 fc 00 00 00 00
e: c7 45 f8 01 00 00 00
15: eb 0a
17: 8b 45 f8
1a: 01 45 fc
1d: 83 45 f8 01
21: 8b 45 f8
24: 3b 45 ec
27: 7e ee
29: 8b 45 fc
2c: 5d
2d: c3
```

a.o

```
push    %rbp
mov     %rsp,%rbp
mov     %edi,-0x14(%rbp)
movl    $0x0,-0x4(%rbp)
movl    $0x1,-0x8(%rbp)
jmp     21 <foo+0x21>
mov     -0x8(%rbp),%eax
add     %eax,-0x4(%rbp)
addl    $0x1,-0x8(%rbp)
mov     -0x8(%rbp),%eax
cmp     -0x14(%rbp),%eax
jle     17 <foo+0x17>
mov     -0x4(%rbp),%eax
pop     %rbp
retq
```

链接



- 将多个二进制目标代码拼接在一起
 - C中称为编译单元 (compilation unit)

a.c

```
int foo(int n) {  
    int sum = 0;  
    for (int i = 1; i <= n; i++) {  
        sum += i;  
    }  
    return sum;  
}
```

gcc -c a.c

a.o

b.c

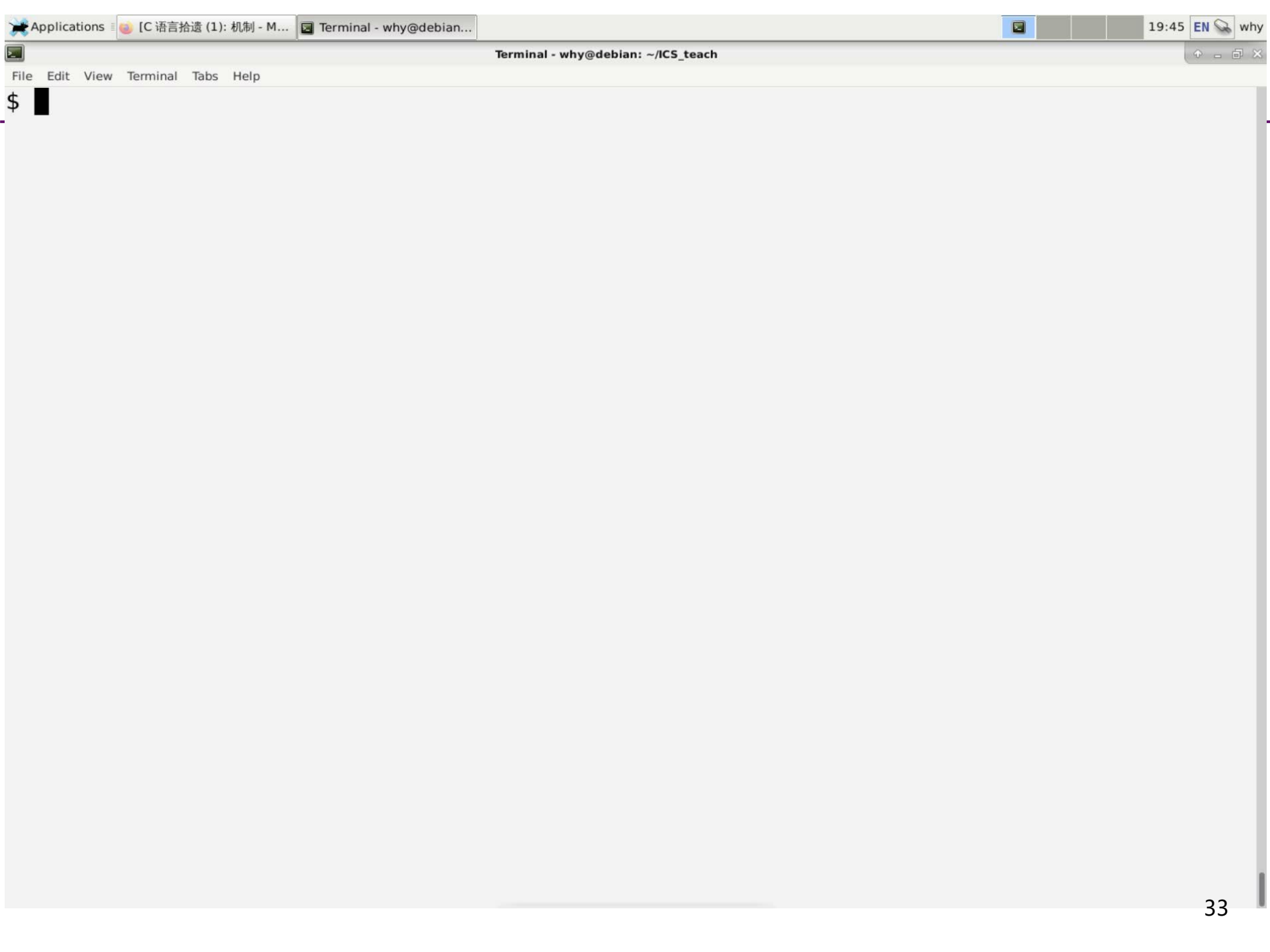
```
#include <stdio.h>  
int foo(int n);  
int main(){  
    printf("%d\n"),foo(100));  
}
```

gcc -c b.c

b.o

链接

?



链接

- 将多个二进制目标代码拼接在一起
 - C中称为编译单元 (compilation unit)
 - 甚至可以链接C++, rust, ...代码

```
extern "C" {  
    int foo() { return 0; }  
}
```

```
int bar() { return 0; }
```

```
$ g++ -c a.cc
```

```
$ objdump -d a.o
```

```
a.o:      file format elf64-x86-64
```

```
Disassembly of section .text:
```

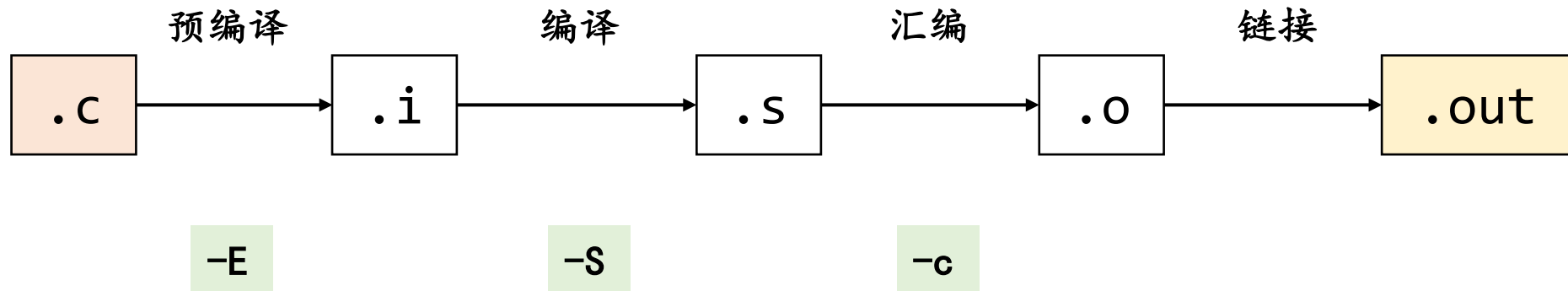
```
0000000000000000 <foo>:
```

0:	55	push	%rbp
1:	48 89 e5	mov	%rsp,%rbp
4:	b8 00 00 00 00	mov	\$0x0,%eax
9:	5d	pop	%rbp
a:	c3	retq	

```
000000000000000b <_Z3barv>:
```

b:	55	push	%rbp
c:	48 89 e5	mov	%rsp,%rbp
f:	b8 00 00 00 00	mov	\$0x0,%eax
14:	5d	pop	%rbp
15:	c3	retq	

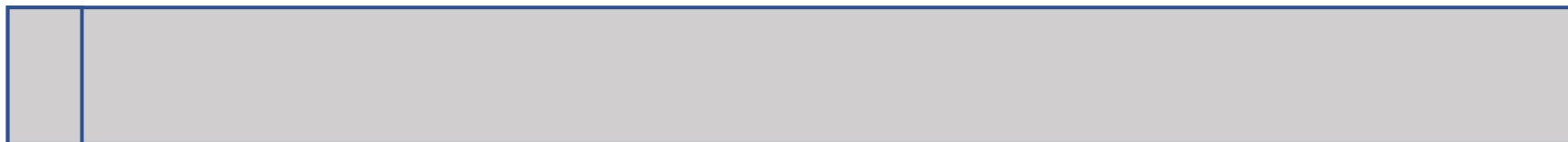
从源代码到可执行文件



加载：进入C语言的世界

C程序执行的两个视角

- 静态：C 代码的连续一段总能对应到一段连续的机器指令
- 动态：C 代码执行的状态总能对应到机器的状态
 - 源代码视角
 - 函数、变量、指针.....
 - 机器指令视角
 - 寄存器、内存、地址.....
- 两个视角的共同之处：内存
 - 代码、变量 (源代码视角) = 地址 + 长度 (机器指令视角)
 - (不太严谨地) 内存 = 代码 + 数据 + 堆栈
 - 因此理解 C 程序执行最重要的就是内存模型



$x=1 \rightarrow$

```
*p = &x
*p = 1
```

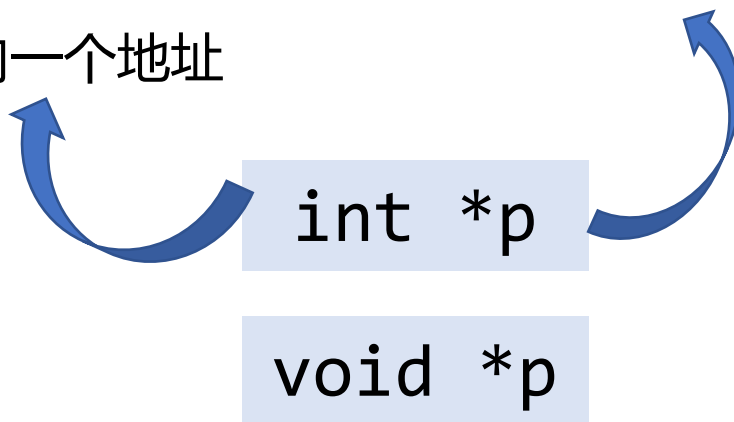
```
int main(int argc, char *argv[]) {
    int *p = (void *) 1;    //OK
    *p = 1;    //Segmentation fault
}
```

内存



对指向的内存的解读: **int**

指针: 指向一个地址



内存

对a地址开始的内存的解读: long



取此块内存的值

`* (long *)`

p指针: 指向一个地址a

`%p`输出地址a

`long *`

`void *p`

```
void printptr(void *p) {  
    printf("p = %p; *p = %016lx\n", p, *(long *)p);  
}
```

指向的内存解读为long输出

输出指针指向的地址

以16位16进制数格式输出: $16 \times 4 = 64\text{bit}$

main, argc和argv

- 一切皆可取地址!

```
void printptr(void *p) {  
    printf("p = %p; *p = %016lx\n", p, *(long *)p);  
}  
int x;  
int main(int argc, char *argv[]) {  
    printptr(main); // 代码  
    printptr(&main);  
    printptr(&x); // 数据  
    printptr(&argc); // 堆栈  
    printptr(argv);  
    printptr(&argv);  
    printptr(argv[0]);  
}
```

指向的地址处内存解读为long输出16位16进制

输出指针指向的地址



\$ █



C Type System

- **类型**：对一段内存的**解读方式**
 - 非常汇编：没有class, polymorphism, type traits,
 - C里的所有的数据都可以理解成**地址（指针）+类型（对地址的解读）**
- 例子（是不是感到学到了假了C语言）

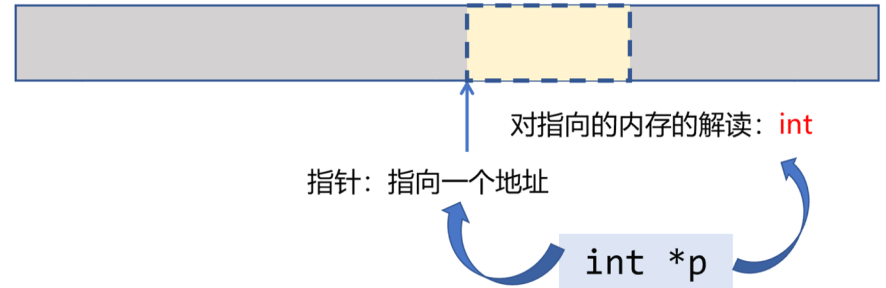
```
int main(int argc, char *argv[]) {  
    int (*f)(int, char *[]) = main;  
    if (argc != 0) {  
        char ***a = &argv, *first = argv[0], ch = argv[0][0];  
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);  
        assert(**a == ch);  
        f(argc - 1, argv + 1);  
    }  
}
```

```
$ ./a.out 1 2 3 hello  
arg = "./a.out"; ch = '.'  
arg = "1"; ch = '1'  
arg = "2"; ch = '2'  
arg = "3"; ch = '3'  
arg = "hello"; ch = 'h'
```



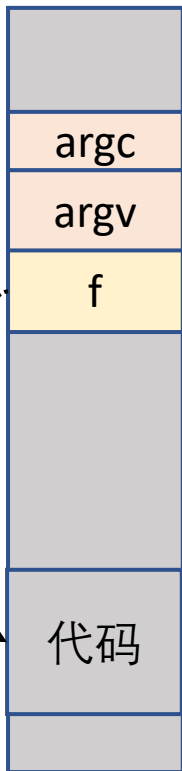
```
int main(int argc, char *argv[]) {  
    int (*f)(int, char *[]) = main;  
    if (argc != 0) {  
        char ***a = &argv, *first = argv[0], ch = argv[0][0];  
        printf("arg = \"%s\""; ch  
        assert(**a == ch);  
        f(argc - 1, argv + 1);  
    }  
}
```

函数指针f



内存

堆栈



char *argv[] → char **argv → (char *)*argv

4字节

? 指针, 存储地址, 64位机器, 8字节

? 指针, 存储地址, 64位机器, 8字节

main

代码



```
int main(int argc, char *argv[]) {  
    int (*f)(int, char *[]) = main;  
    if (argc != 0) {  
        char ***a = &argv, *first  
        printf("arg = \"%s\""; ch  
        assert(**a == ch);  
        f(argc - 1, argv + 1);  
    }  
}
```



对指向的内存的解读: int

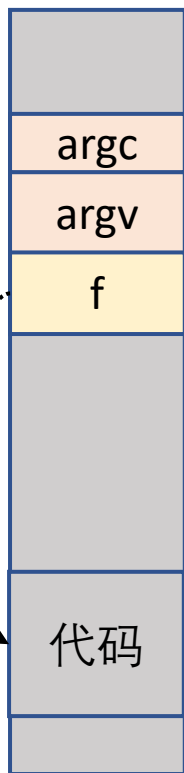
指针: 指向一个地址

int *p

char *argv[] → char **argv → (char *)*argv

内存

堆栈



4字节

8字节

8字节

解读为char *

解读为char

地址, 8字节

1字节

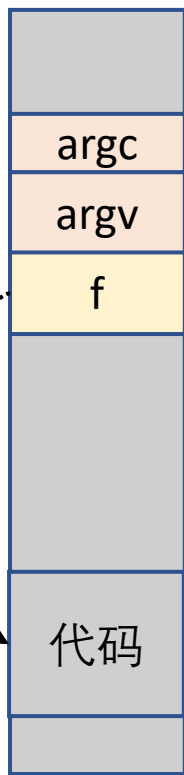


```
int main(int argc, char *argv[]) {  
    int (*f)(int, char *[]) = main;  
    if (argc != 0) {  
        char ***a = &argv, *first = argv[0], ch = argv[0][0];  
        printf("arg = \"%s\"; ch = '%c'\\n", first, ch);  
        assert(**a == ch);  
        f(argc - 1, argv + 1);  
    }  
}
```

内存

堆栈

main



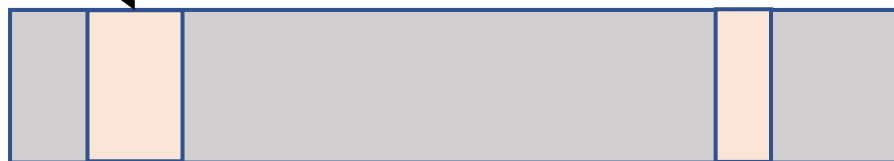
4字节

8字节

8字节

代码

解读为char *



地址, 8字节

1字节

解读为char

```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```

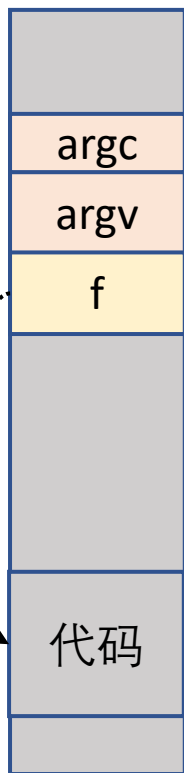


什么是argv+1?

内存

堆栈

main



4字节

8字节

8字节

解读为char *

地址, 8字节

1字节

解读为char

int *p; → p+1
按int解读下一个内存地址

```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

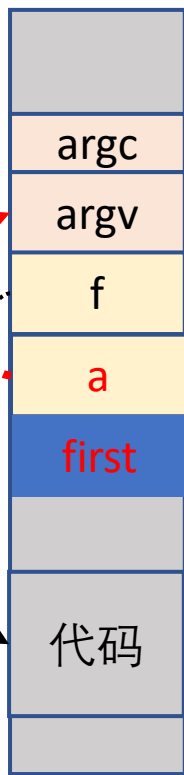
```



内存

堆栈

main



4字节

8字节

8字节

8字节

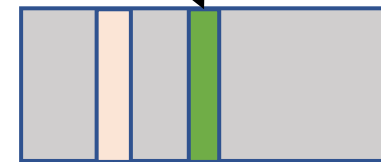
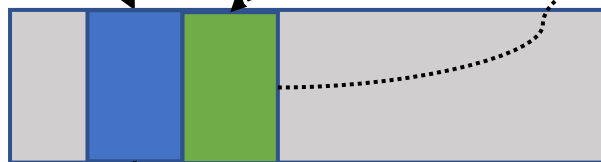
8字节

地址, 8字节

按char解读

argv+1、
&argv[1]

解读为char *



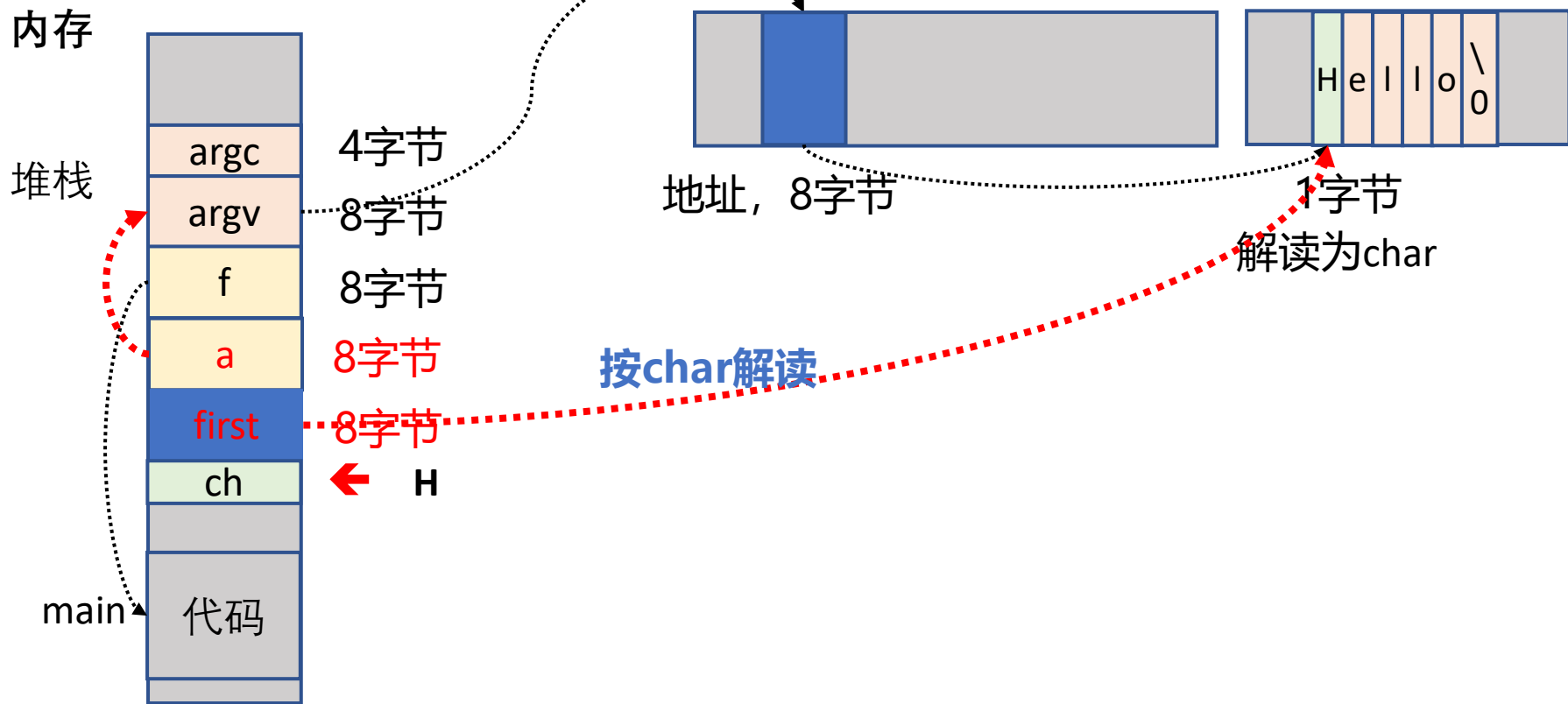
1字节

解读为char


```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

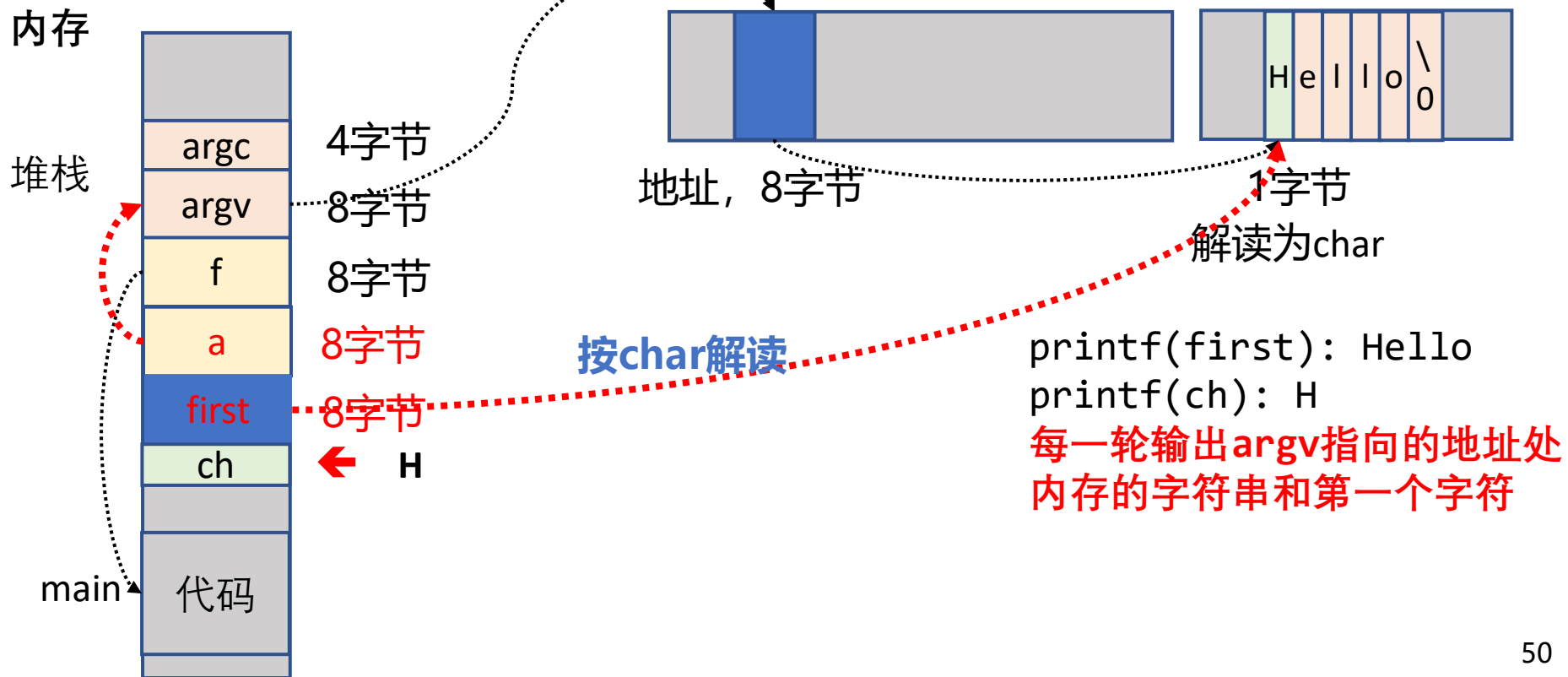
```



```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\\n", first, ch);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```



```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0];
        printf("arg = \"%s\"; ch = '%c'\\n", *first, **a);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```

↓

```

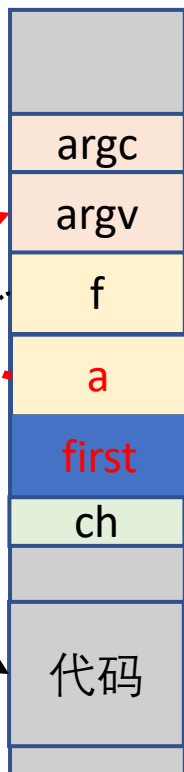
$ ./a.out 1 2 3 hello
arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'

```

内存

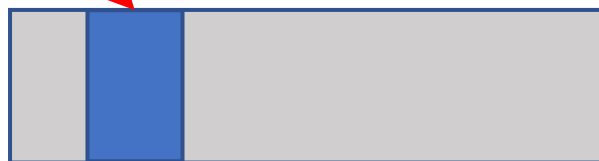
堆栈

main

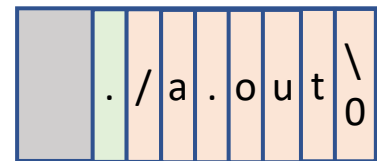


5

解读为char *



Hello



地址, 8字节

1字节

解读为char

按char解读

每一轮输出argv指向的地址所存地址处内存的字符串和第一个字符

arg = "./a.out"; ch = '.'

./a.out 1 2 3 Hello

```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0];
        printf("arg = \"%s\"; ch = '%c'\\n", *first, **a);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```



```

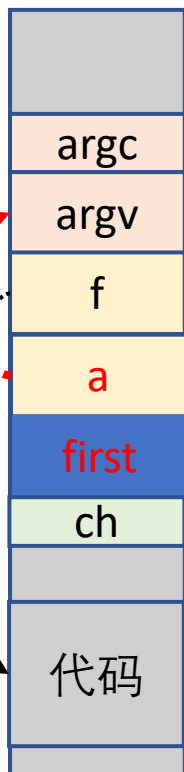
$ ./a.out 1 2 3 hello
arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'

```

内存

堆栈

main



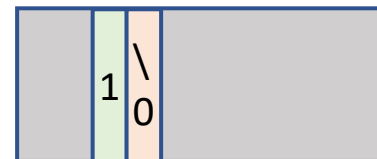
4

解读为char *



地址, 8字节

Hello



1字节

解读为char

按char解读

每一轮输出argv指向的地址所存地址处内存的字符串和第一个字符

arg = "./a.out"; ch = '.'

arg = "1"; ch = '1'

./a.out 1 2 3 Hello

```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0];
        printf("arg = \"%s\"; ch = '%c'\\n", *first, **a);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```

↓

```

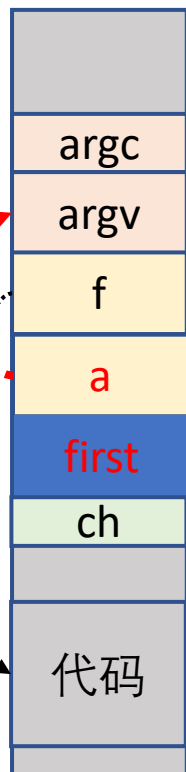
$ ./a.out 1 2 3 hello
arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'

```

内存

堆栈

main



3

地址, 8字节

按char解读

解读为char *

Hello

1字节

解读为char

每一轮输出argv指向的地址所存地址处内存的字符串和第一个字符

arg = "./a.out"; ch = '.'

arg = "1"; ch = '1'

arg = "2"; ch = '2'

./a.out 1 2 3 Hello

```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0];
        printf("arg = \"%s\"; ch = '%c'\\n", *first, **a);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```

```

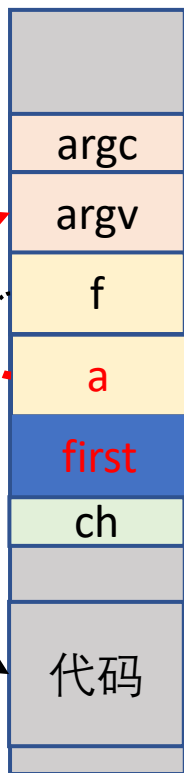
$ ./a.out 1 2 3 hello
arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'

```

内存

堆栈

main



2

地址, 8字节

按char解读

解读为char*

Hello

1字节

解读为char

每一轮输出argv指向的地址所存地址处内存的字符串和第一个字符

```

arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'

```

./a.out 1 2 3 Hello

```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0];
        printf("arg = \"%s\"; ch = '%c'\\n", *first, **a);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```

↓

```

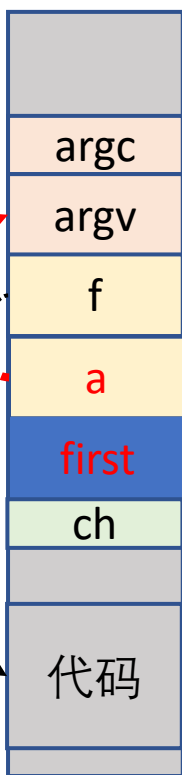
$ ./a.out 1 2 3 hello
arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'

```

内存

堆栈

main



1

按char解读

解读为char*

地址, 8字节

Hello

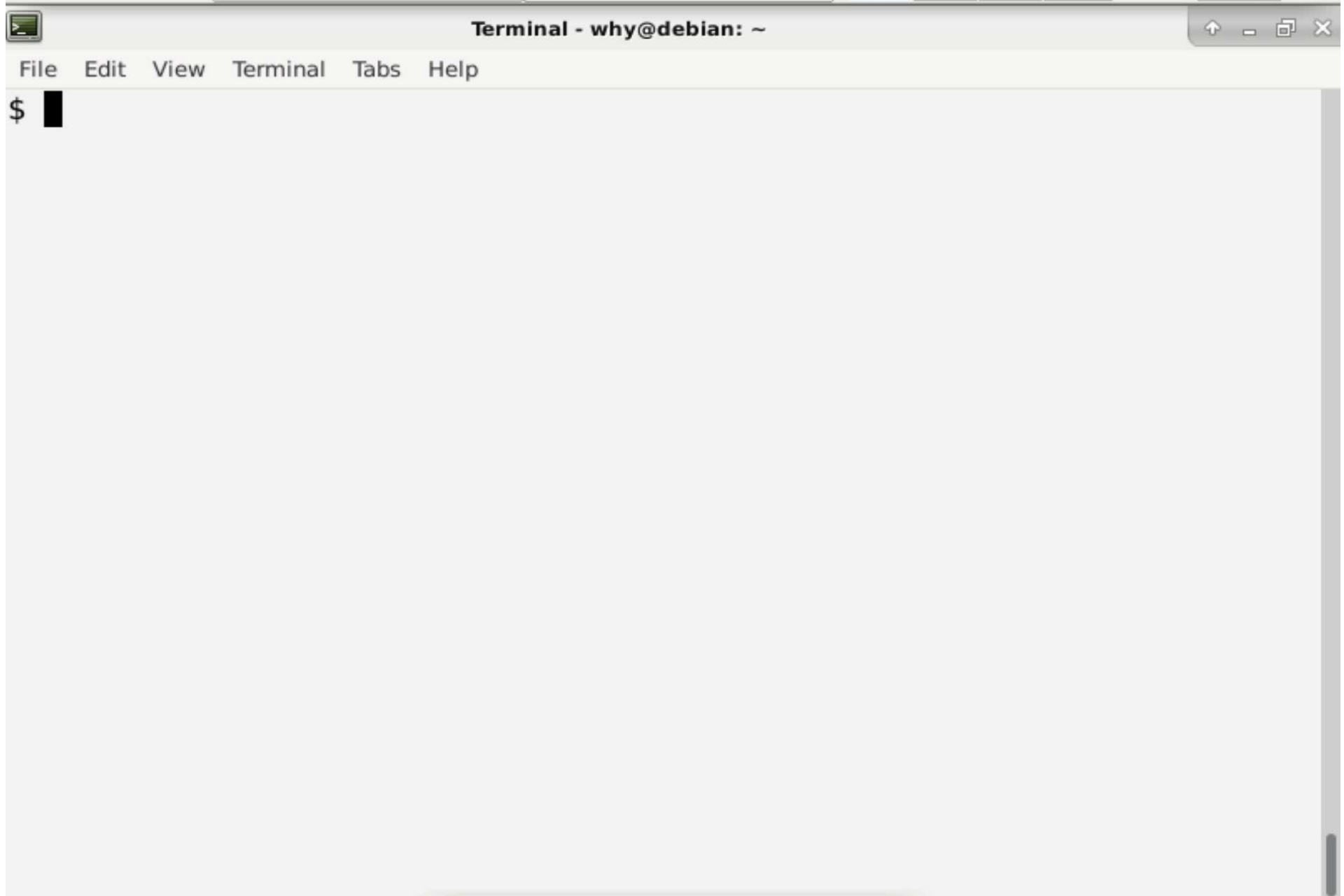
1字节

解读为char

每一轮输出argv指向的地址所存地址处内存的字符串和第一个字符

./a.out 1 2 3 Hello

arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'




```
void printptr(void *p) { 指向的地址处内存解读为long输出16位16进制
    printf("p = %p; *p = %016lx\n", p, *(long *)p);
}
```

输出指针指向的地址

```
int x;
int main(int argc, char *argv[]) {
    printptr(main); // 代码
    printptr(&main);
    printptr(&x); // 数据
    printptr(&argc); // 堆栈
    printptr(argv);
    printptr(&argv);
    printptr(argv[0]);
}
```

```
$ ./a.out
```

```
p = 0x563afd3cf163; *p = 10ec8348e5894855
p = 0x563afd3cf163; *
p = 0x563afd3d2034; *
p = 0x7ffcada0761c; *p = fd3cf1d000000001
p = 0x7ffcada07708; *p = 00007ffcada084fe
p = 0x7ffcada07610; *p = 00007ffcada07708
p = 0x7ffcada084fe; *p = 0074756f2e612f2e
```

地址XX, 地址XX内存的内容

内存

堆栈
a→
b→



4字节
8字节

地址, 8字节

d

解读为char

	p	*p
printptr(&argc)	a	argc值
printptr(argv)	c	d
printptr(&argv)	b	c
printptr(argv[0])	d	d处字符串

从main函数开始执行

- 标准规定C程序从main开始执行
 - （思考题：谁调用的main？ 进程执行的第一条指令是什么？）

```
int main(int argc, char *argv[]);
```

- argc (argument count): 参数个数
- argv (argument vector): 参数列表 (NULL结束)
- `ls -al`
 - `argc = 2, argv = ["ls", "-al", NULL]`

End.

- C语言简单（在可控时间成本里可以精通）
- C语言通用（大量系统是C语言编写的）
- C语言实现对底层机器的精准控制（鸿蒙）
- 推荐阅读：[The Art of Readable Code](#)