C语言拾遗：机制和实践

王慧妍

why@nju.edu.cn

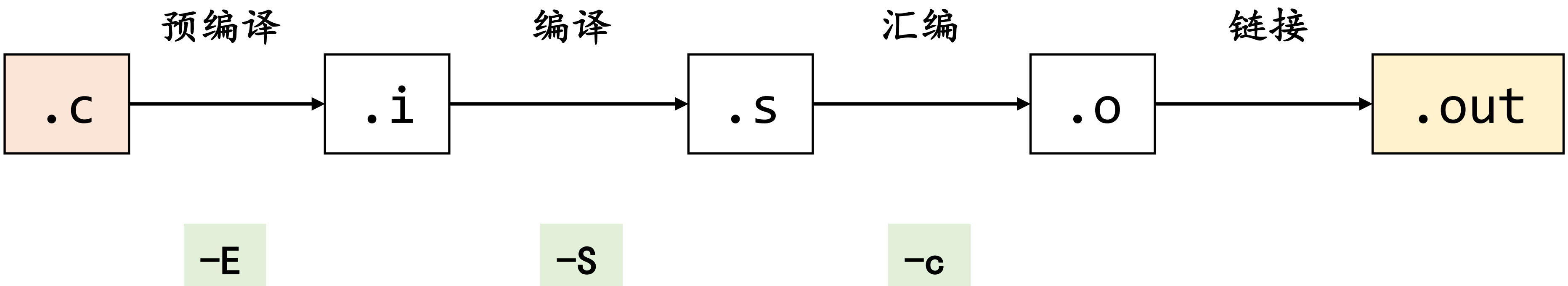
南京大学　　软件学院　　计算机软件研究所

讲前提醒

Token已发送，PA0截止时间：
* 2025年9月21日23:59:59 (以此 deadline 计按时提交bonus)

PA1已悄悄发布：（OJ已开启PA1评测）
* PA 1.1: 2025.10.5 (此为建议的不计分 deadline)
* PA 1.2: 2025.10.12 (此为建议的不计分 deadline)
* PA 1.3: 2025.10.19 23:59:59 (以此 deadline 计按时100%提交)

从源代码到可执行文件



预编译

（先行剧透本学期的主要内容）

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

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

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

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

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

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

X-Macros

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

```c
#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)
```

PA中有趣的预编译

- PA头文件遵循标准头文件结构，避免被重复引用
  - in nemu/include/isa.h

```
#ifndef __ISA_H__
#define __ISA_H__
......
#endif
```

  - in nemu/include/common.h
    - 实际上是include哪一些头文件？
    - CONFIG_TARGET_AM是否有定义，在哪里?
      - make menuconfig做了什么？

```
#ifdef CONFIG_TARGET_AM
#include <klib.h>
#else
#include <assert.h>
#include <stdlib.h>
#endif
```

PA中有趣的预编译

```c
#ifdef __cplusplus
extern "C" {
#endif

// ----------------------- TRM: Turing Machine -----------------------
extern  Area        heap;
void    putch       (char ch);
void    halt        (int code) __attribute__((__noreturn__));

// -------------------- IOE: Input/Output Devices --------------------
bool    ioe_init    (void);
void    ioe_read    (int reg, void *buf);
void    ioe_write   (int reg, void *buf);
#include "amdev.h"

// ---------- CTE: Interrupt Handling and Context Switching ----------
bool    cte_init    (Context *(*handler)(Event ev, Context *ctx));
void    yield       (void);
bool    ienabled    (void);
void    iset        (bool enable);
Context *kcontext   (Area kstack, void (*entry)(void *), void *arg);

// ----------------------- VME: Virtual Memory -----------------------
bool    vme_init    (void *(*pgalloc)(int), void (*pgfree)(void *));
void    protect     (AddrSpace *as);
void    unprotect   (AddrSpace *as);
void    map         (AddrSpace *as, void *vaddr, void *paddr, int prot);
Context *ucontext   (AddrSpace *as, Area kstack, void *entry);

// ---------------------- MPE: Multi-Processing ----------------------
bool    mpe_init    (void (*entry)());
int     cpu_count   (void);
int     cpu_current (void);
int     atomic_xchg (int *addr, int newval);

#ifdef __cplusplus
}
#endif
```

in abstract-machine/am/include/am.h

PA中有趣的预编译

```c
#define _Log(...) \
  do { \
    printf(__VA_ARGS__); \
    log_write(__VA_ARGS__); \
  } while (0)
```

为什么要用do{...} while(0)？

```c
#if !defined(likely)
#define likely(cond)   __builtin_expect(cond, 1)
#define unlikely(cond) __builtin_expect(cond, 0)
#endif
```

likely和unlikely？

```c
#define ANSI_FG_BLACK   "\33[1;30m"
#define ANSI_FG_RED     "\33[1;31m"
#define ANSI_FG_GREEN   "\33[1;32m"
#define ANSI_FG_YELLOW  "\33[1;33m"
#define ANSI_FG_BLUE    "\33[1;34m"
#define ANSI_FG_MAGENTA "\33[1;35m"
#define ANSI_FG_CYAN    "\33[1;36m"
#define ANSI_FG_WHITE   "\33[1;37m"
#define ANSI_BG_BLACK   "\33[1;40m"
#define ANSI_BG_RED     "\33[1;41m"
#define ANSI_BG_GREEN   "\33[1;42m"
#define ANSI_BG_YELLOW  "\33[1;43m"
#define ANSI_BG_BLUE    "\33[1;44m"
#define ANSI_BG_MAGENTA "\33[1;45m"
#define ANSI_BG_CYAN    "\33[1;46m"
#define ANSI_BG_WHITE   "\33[1;47m"
#define ANSI_NONE       "\33[0m"
```

这个是什么？

PA中有趣的预编译

- 宏维护按键名称列表

```c
#define AM_KEYS(_) \
  _(ESCAPE) _(F1) _(F2) _(F3) _(F4) _(F5) _(F6) _(F7) _(F8) _(F9) _(F10) _(F11) _(F12) \
  _(GRAVE) _(1) _(2) _(3) _(4) _(5) _(6) _(7) _(8) _(9) _(0) _(MINUS) _(EQUALS) _(BACKSPACE) \
  _(TAB) _(Q) _(W) _(E) _(R) _(T) _(Y) _(U) _(I) _(O) _(P) _(LEFTBRACKET) _(RIGHTBRACKET) _(BACKSLASH) \
  _(CAPSLOCK) _(A) _(S) _(D) _(F) _(G) _(H) _(J) _(K) _(L) _(SEMICOLON) _(APOSTROPHE) _(RETURN) \
  _(LSHIFT) _(Z) _(X) _(C) _(V) _(B) _(N) _(M) _(COMMA) _(PERIOD) _(SLASH) _(RSHIFT) \
  _(LCTRL) _(APPLICATION) _(LALT) _(SPACE) _(RALT) _(RCTRL) \
  _(UP) _(DOWN) _(LEFT) _(RIGHT) _(INSERT) _(DELETE) _(HOME) _(END) _(PAGEUP) _(PAGEDOWN)

#define AM_KEY_NAMES(key) AM_KEY_##key,
enum {
  AM_KEY_NONE = 0,
  AM_KEYS(AM_KEY_NAMES)
};
```

可以使用类似的宏技巧来批量定义变量。以下是几种常见的写法:

1. 批量定义整型变量

```c
#define DEFINE_INT_VARS(_) \
    _(var1) _(var2) _(var3) _(var4) _(var5)

#define DECLARE_INT(var) int var;

// 使用方式
DEFINE_INT_VARS(DECLARE_INT)
// 展开为: int var1; int var2; int var3; int var4; int var5;
```

PA中有趣的预编译

- 阅读PA代码的第一个困难(in nemu/src/nemu-main.c)
  - am_init_monitor()?
  - 还是init_monitor()?

```
int main(int argc, char*argv[]) {
  /* Initialize the monitor. */
#ifdef CONFIG_TARGET_AM
  am_init_monitor();
#else
  init_monitor(argc, argv);
#endif
  /* Start engine. */
  engine_start();
  returnis_exit_status_bad();
}
```

PA1阶段相关的宏

- in nemu/src/monitor/monitor.c

```c
#ifndef CONFIG_TARGET_AM
#include <getopt.h>

void sdb_set_batch_mode();

static char *log_file = NULL;
static char *diff_so_file = NULL;
static char *img_file = NULL;
static int difftest_port = 1234;

static long load_img() {…

static int parse_args(int argc, char *argv[]) {…

void init_monitor(int argc, char *argv[]) {…
#else // CONFIG_TARGET_AM…
#endif
```

```c
/* Initialize the simple debugger. */
init_sdb();

IFDEF(CONFIG_ITRACE, init_disasm());
```

PA2阶段最重要的宏

- 看看能不能读懂？
  - 可变长宏参数（ ）

```c
// --- pattern matching wrappers for decode ---
#define INSTPAT(pattern, ...) do { \
  uint64_t key, mask, shift; \
  pattern_decode(pattern, STRLEN(pattern), &key, &mask, &shift); \
  if ((((uint64_t)INSTPAT_INST(s) >> shift) & mask) == key) { \
    INSTPAT_MATCH(s, ##__VA_ARGS__); \
    goto *(__instpat_end); \
  } \
} while (0)

#define INSTPAT_START(name) { const void * __instpat_end = &&concat(__instpat_end_, name);
#define INSTPAT_END(name)   concat(__instpat_end_, name): ; }
```

PA中丰富可用的宏

cpu.h ● macro.h 2 ×

ics2025 > nemu > include > macro.h > MAP(c, f)

```c
#ifndef __MACRO_H__

// calculate the length of an array
#define ARRLEN(arr) (int)(sizeof(arr) / sizeof(arr[0]))

// macro concatenation
#define concat_temp(x, y) x ## y
#define concat(x, y) concat_temp(x, y)
#define concat3(x, y, z) concat(concat(x, y),
#define concat4(x, y, z, w) concat3(concat(x,
#define concat5(x, y, z, v, w) concat4(concat(

// macro testing
// See https://stackoverflow.com/questions/26
#define CHOOSE2nd(a, b, ...) b
#define MUX_WITH_COMMA(contain_comma, a, b) CH
#define MUX_MACRO_PROPERTY(p, macro, a, b) MU
// define placeholders for some property
#define __P_DEF_0  X,
#define __P_DEF_1  X,
#define __P_ONE_1  X,
#define __P_ZERO_0 X,
// define some selection functions based on th
#define MUXDEF(macro, X, Y)  MUX_MACRO_PROPERT
#define MUXNDEF(macro, X, Y) MUX_MACRO_PROPERT
#define MUXONE(macro, X, Y)  MUX_MACRO_PROPERT
#define MUXZERO(macro, X, Y) MUX_MACRO_PROPERT

// test if a boolean macro is defined
#define ISDEF(macro) MUXDEF(macro, 1, 0)
// test if a boolean macro is undefined
#define ISNDEF(macro) MUXNDEF(macro, 1, 0)
// test if a boolean macro is defined to 1
#define ISONE(macro) MUXONE(macro, 1, 0)
// test if a boolean macro is defined to 0
#define ISZERO(macro) MUXZERO(macro, 1, 0)
// test if a macro of ANY type is defined
// NOTE1: it ONLY works inside a function, since it calls `strcmp()`
// NOTE2: macros defined to themselves (#define A A) will get wrong results
#define isdef(macro) (strcmp("" #macro, "" str(macro)) != 0)
```

```c
#define Log(format, ...) \
    _Log(ANSI_FMT("[%s:%d %s] " format, ANSI_FG_BLUE) "\n", \
        __FILE__, __LINE__, __func__, ## __VA_ARGS__)

#define Assert(cond, format, ...) \
  do { \
    if (!(cond)) { \
      MUXDEF(CONFIG_TARGET_AM, printf(ANSI_FMT(format, ANSI_FG_RED) "\n", ## __VA_ARGS__), \
        (fflush(stdout), fprintf(stderr, ANSI_FMT(format, ANSI_FG_RED) "\n", ##  __VA_ARGS__))); \
      IFNDEF(CONFIG_TARGET_AM, extern FILE* log_fp; fflush(log_fp)); \
      extern void assert_fail_msg(); \
      assert_fail_msg(); \
      assert(cond); \
    } \
  } while (0)

#define panic(format, ...) Assert(0, format, ## __VA_ARGS__)

#define TODO() panic("please implement me")
```

多路线支持的背后

- ISA_H
  - CFLAGS维护，影响不同路线选择（有兴趣去看看构建过程？）

```c
#if defined(CONFIG_ISA_x86)
# define DIFFTEST_REG_SIZE (sizeof(uint32_t) * 9) // GPRs + pc
#elif defined(CONFIG_ISA_mips32)
# define DIFFTEST_REG_SIZE (sizeof(uint32_t) * 38) // GPRs + status + lo + hi + badvaddr
+ cause + pc
#elif defined(CONFIG_ISA_riscv)
#define RISCV_GPR_TYPE MUXDEF(CONFIG_RV64, uint64_t, uint32_t)
#define RISCV_GPR_NUM  MUXDEF(CONFIG_RVE , 16, 32)
#define DIFFTEST_REG_SIZE (sizeof(RISCV_GPR_TYPE) * (RISCV_GPR_NUM
#elif defined(CONFIG_ISA_loongarch32r)
# define DIFFTEST_REG_SIZE (sizeof(uint32_t) * 33) // GPRs + pc
#else
# error Unsupport ISA
#endif
```

```c
#if defined(__ARCH_X86_NEMU)
# define DEVICE_BASE 0x0
#else
# define DEVICE_BASE 0xa0000000
#endif

#define MMIO_BASE 0xa0000000

#define SERIAL_PORT     (DEVICE_BASE + 0x00003f8)
#define KBD_ADDR        (DEVICE_BASE + 0x0000060)
#define RTC_ADDR        (DEVICE_BASE + 0x0000048)
#define VGACTL_ADDR     (DEVICE_BASE + 0x0000100)
#define AUDIO_ADDR      (DEVICE_BASE + 0x0000200)
#define DISK_ADDR       (DEVICE_BASE + 0x0000300)
#define FB_ADDR         (MMIO_BASE   + 0x1000000)
#define AUDIO_SBUF_ADDR (MMIO_BASE   + 0x1200000)
```

```c
#if defined(__ISA_X86__)
# define nemu_trap(code) asm volatile ("int3" : :"a"(code))
#elif defined(__ISA_MIPS32__)
# define nemu_trap(code) asm volatile ("move $v0, %0; sdbbp" :"
#elif defined(__riscv)
# define nemu_trap(code) asm volatile("mv a0, %0; ebreak" : :"
#elif defined(__ISA_LOONGARCH32R__)
# define nemu_trap(code) asm volatile("move $a0, %0; break 0" : :"r"(code))
#else
# error unsupported ISA __ISA__
#endif
```

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）、……

```c
#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！

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

Applications [C 语言拾遗 (1): 机制 - M... Terminal - why@debian... 19:19

Terminal - why@debian: ~/ICS_teach

File Edit View Terminal Tabs Help

```
$
```

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！

```c
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

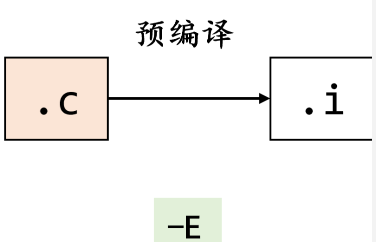
```
foo:
    n = ARG-1
    sum = 0
    i = 1
    goto    .L2
.L3:
    tmp = i
    sum += tmp
    i += 1
.L2:
    tmp = i
    compare (n, tmp)
    if(<=)  goto .L3
    RETURN-VAL = sum

    ret
```

a.c

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

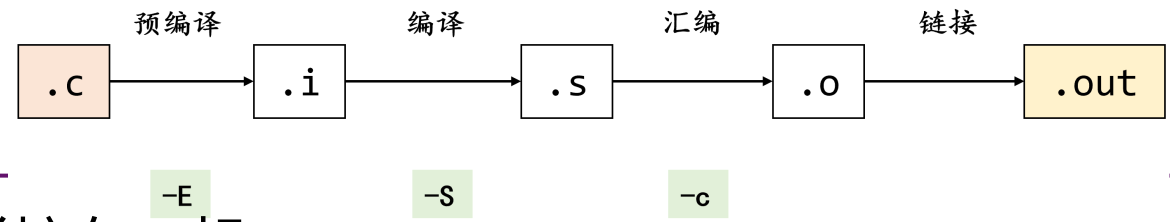
```
$ gcc -c a.c
$ objdump -d a.o

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:   89 7d ec                mov    %edi,-0x14(%rbp)
   7:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
   e:   c7 45 f8 01 00 00 00    movl   $0x1,-0x8(%rbp)
  15:   eb 0a                   jmp    21 <foo+0x21>
  17:   8b 45 f8                mov    -0x8(%rbp),%eax
  1a:   01 45 fc                add    %eax,-0x4(%rbp)
  1d:   83 45 f8 01             addl   $0x1,-0x8(%rbp)
  21:   8b 45 f8                mov    -0x8(%rbp),%eax
  24:   3b 45 ec                cmp    -0x14(%rbp),%eax
  27:   7e ee                   jle    17 <foo+0x17>
  29:   8b 45 fc                mov    -0x4(%rbp),%eax
  2c:   5d                      pop    %rbp
  2d:   c3                      retq
```

a.o



没有main还不能运行

链接



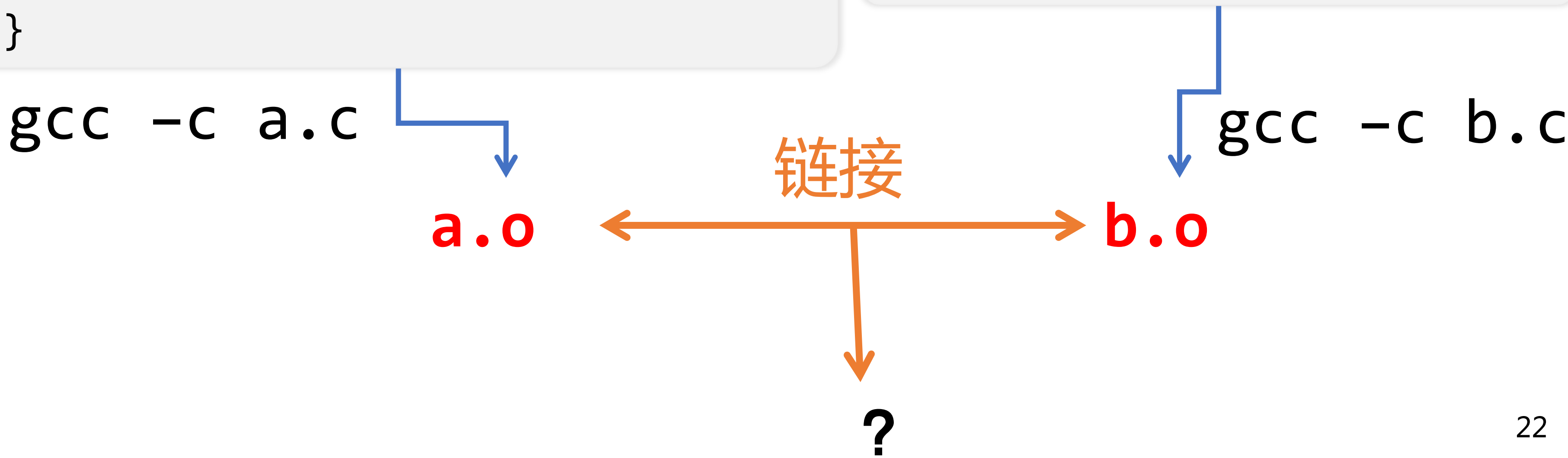
- 将多个二进制目标代码拼接在一起
  - C中称为编译单元（compilation unit）

a.c

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

b.c

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



Applications [C 语言拾遗 (1): 机制 - M... Terminal - why@debian... 19:45 EN

Terminal - why@debian: ~/ICS_teach

File Edit View Terminal Tabs Help

```
$ 
```

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; }
```

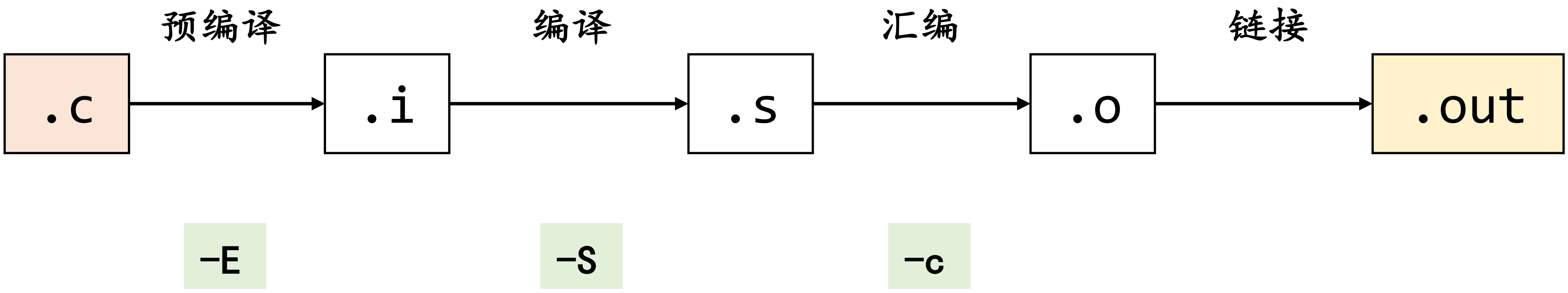
```
$ g++ -c a.cc
$ objdump -d a.o

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
```

PA中的细节

```c
#ifdef __cplusplus
extern "C" {
#endif

// ---------------------- TRM: Turing Machine ----------------------
extern  Area        heap;
void    putch       (char ch);
void    halt        (int code) __attribute__((__noreturn__));

// -------------------- IOE: Input/Output Devices --------------------
bool    ioe_init    (void);
void    ioe_read    (int reg, void *buf);
void    ioe_write   (int reg, void *buf);
#include "amdev.h"

// ---------- CTE: Interrupt Handling and Context Switching ----------
bool    cte_init    (Context *(*handler)(Event ev, Context *ctx));
void    yield       (void);
bool    ienabled    (void);
void    iset        (bool enable);
Context *kcontext   (Area kstack, void (*entry)(void *), void *arg);

// ---------------------- VME: Virtual Memory ----------------------
bool    vme_init    (void *(*pgalloc)(int), void (*pgfree)(void *));
void    protect     (AddrSpace *as);
void    unprotect   (AddrSpace *as);
void    map         (AddrSpace *as, void *vaddr, void *paddr, int prot);
Context *ucontext   (AddrSpace *as, Area kstack, void *entry);

// --------------------- MPE: Multi-Processing ---------------------
bool    mpe_init    (void (*entry)());
int     cpu_count   (void);
int     cpu_current (void);
int     atomic_xchg (int *addr, int newval);

#ifdef __cplusplus
}
#endif
```

in abstract-machine/am/include/am.h

从源代码到可执行文件



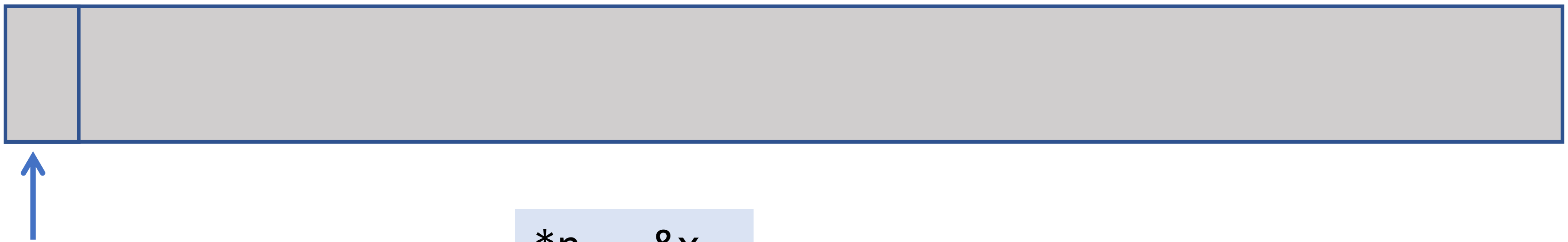
加载：进入C语言的世界

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

内存



x=1 →

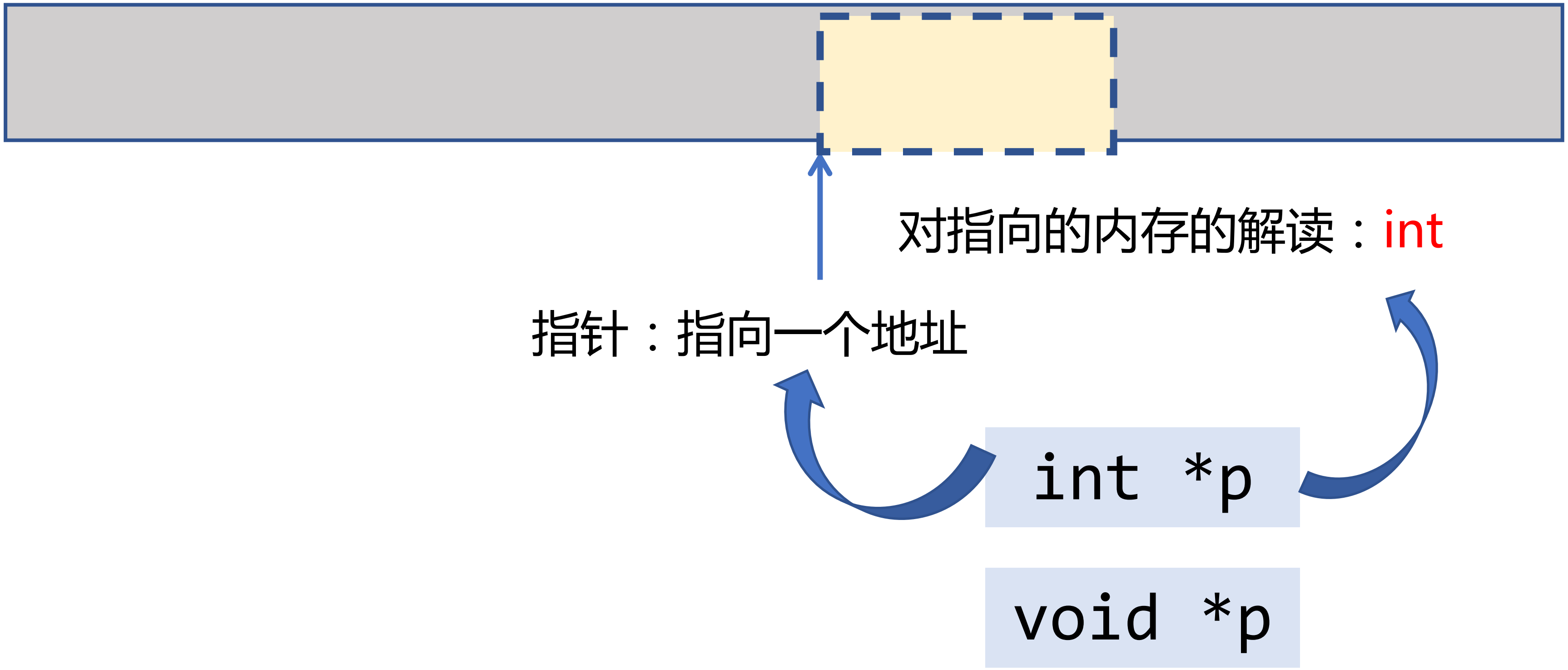
```
*p = &x
*p = 1
```

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

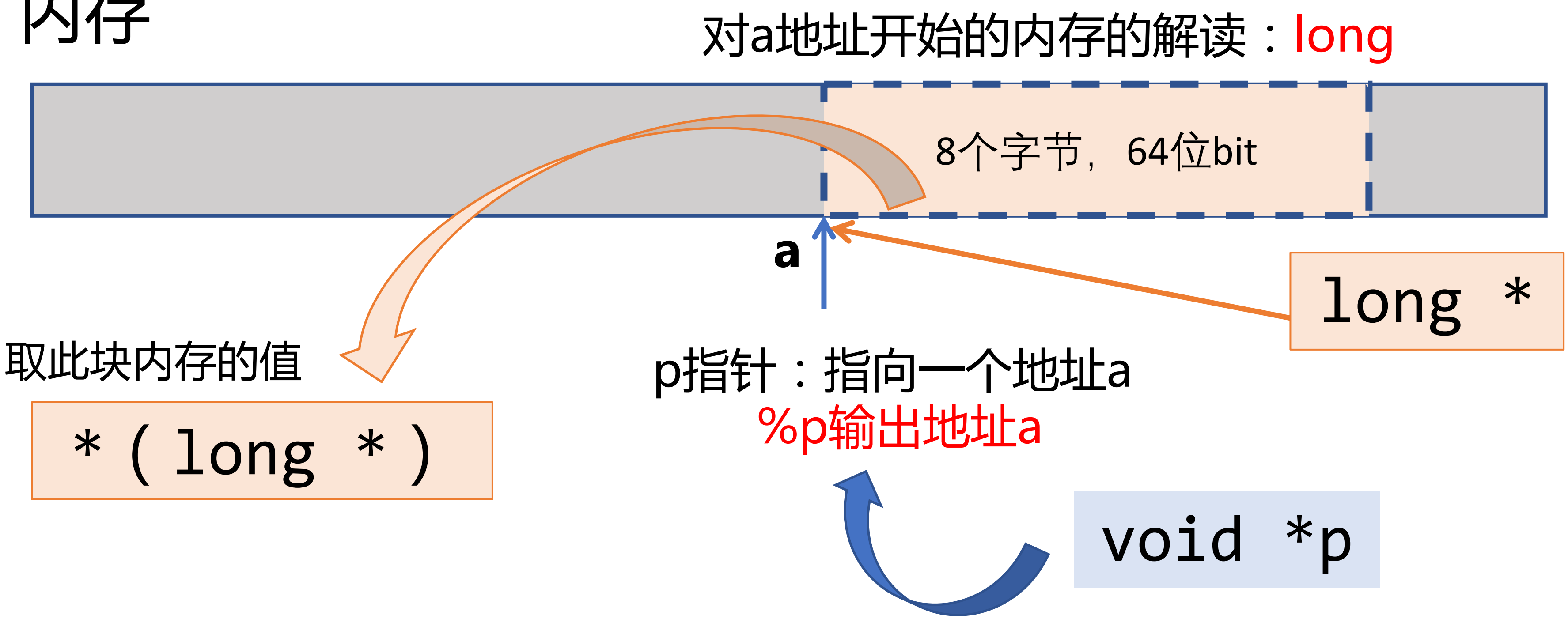
内存只是个**字节序列**

无论何种类型的指针都只是**地址**+对指向内存的**解读**

内存



内存




```c
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*4=64bit

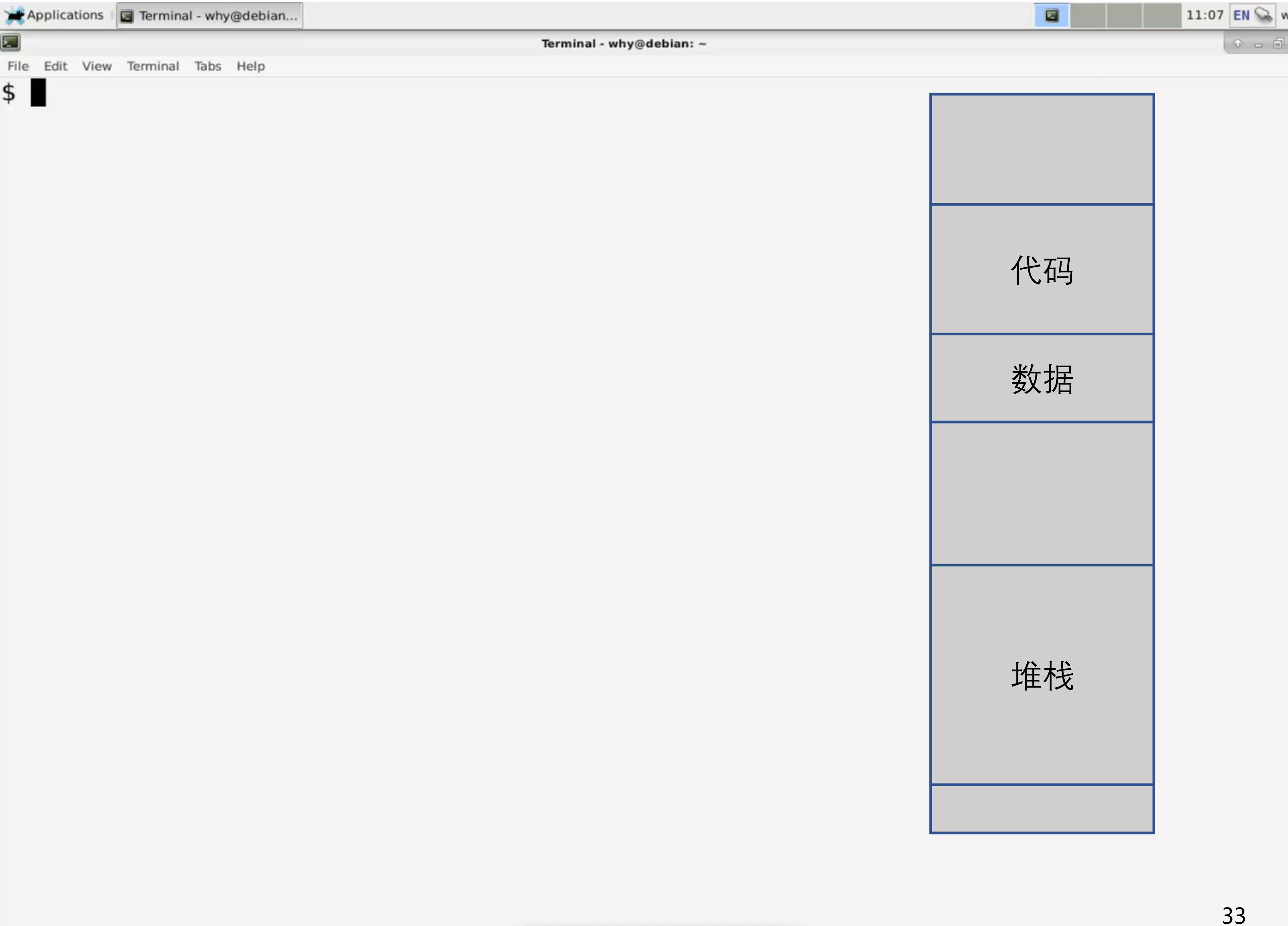
main，argc和argv

- 一切皆可取地址！

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
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]);
}
```


Applications
Terminal - why@debian...
11:07
EN
Terminal - why@debian: ~
File Edit View Terminal Tabs Help
$
代码
数据
堆栈

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）

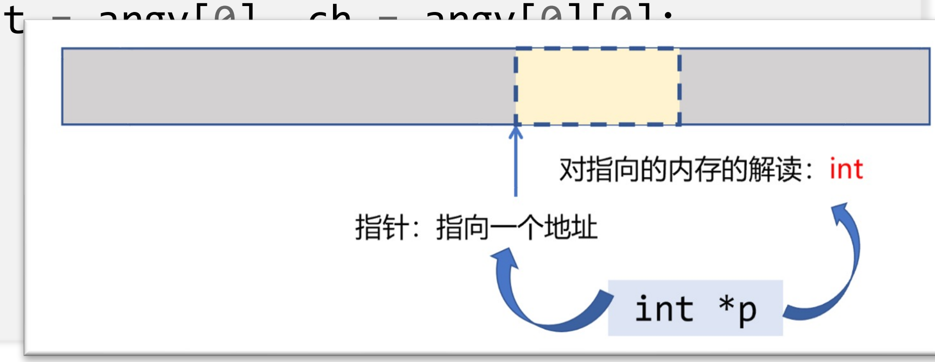
- 例子（是不是感到学到了假了C语言）

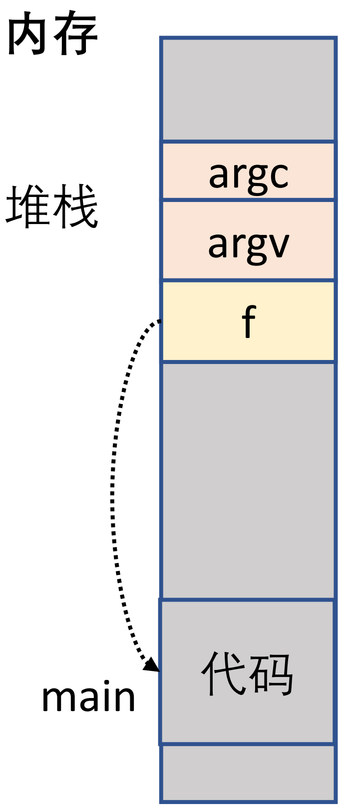
```c
int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(***a == ch);
        f(argc - 1, argv + 1);
    }
}
```

```
$ ./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'
```

```c
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





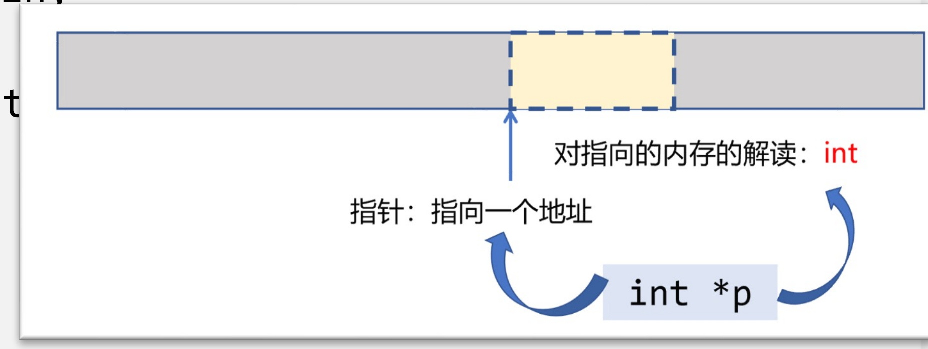
char *argv[] ➔ char **argv ➔ (char *)*argv

4字节

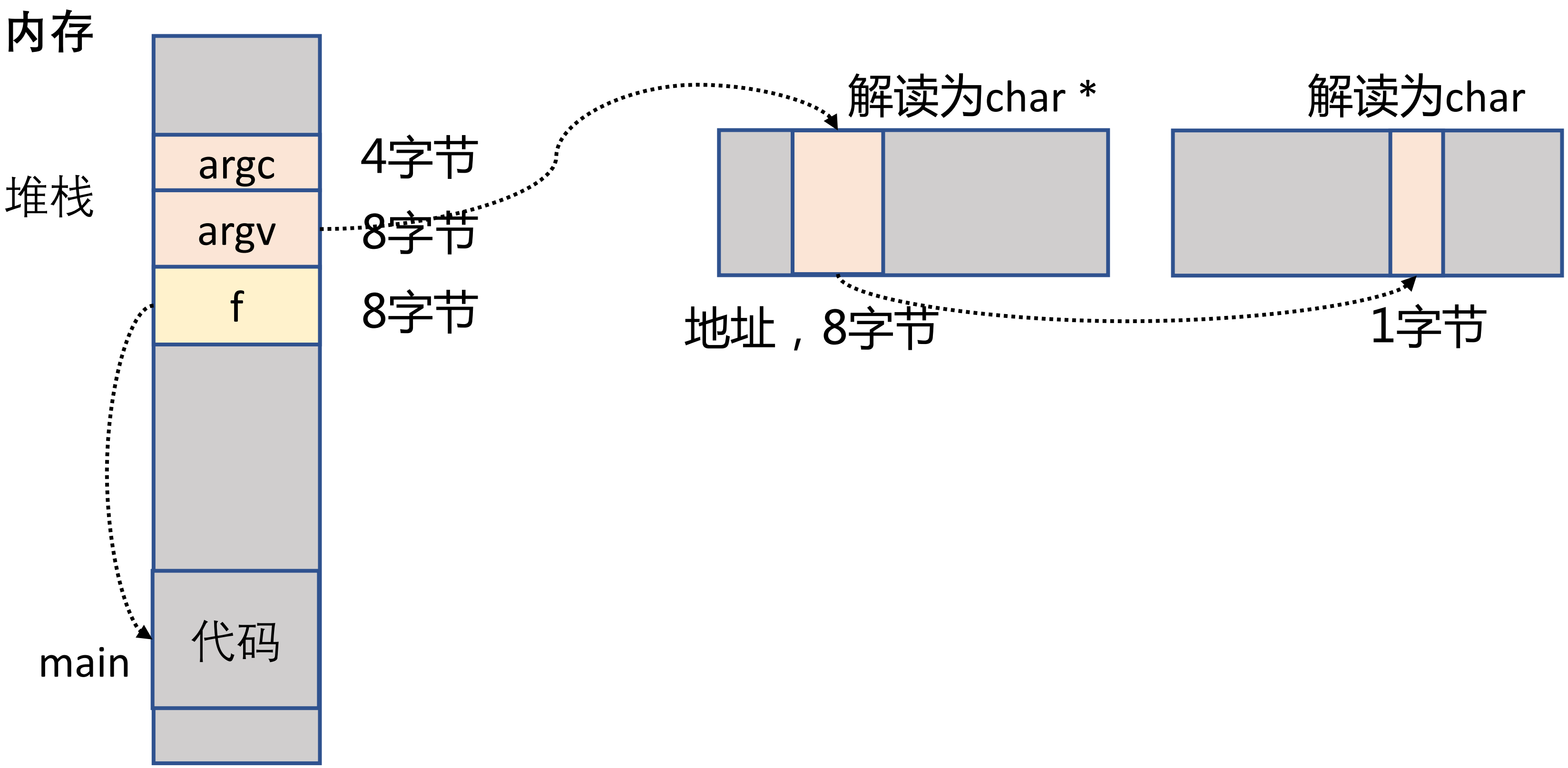
? **指针，存储地址，64位机器，8字节**

? **指针，存储地址，64位机器，8字节**

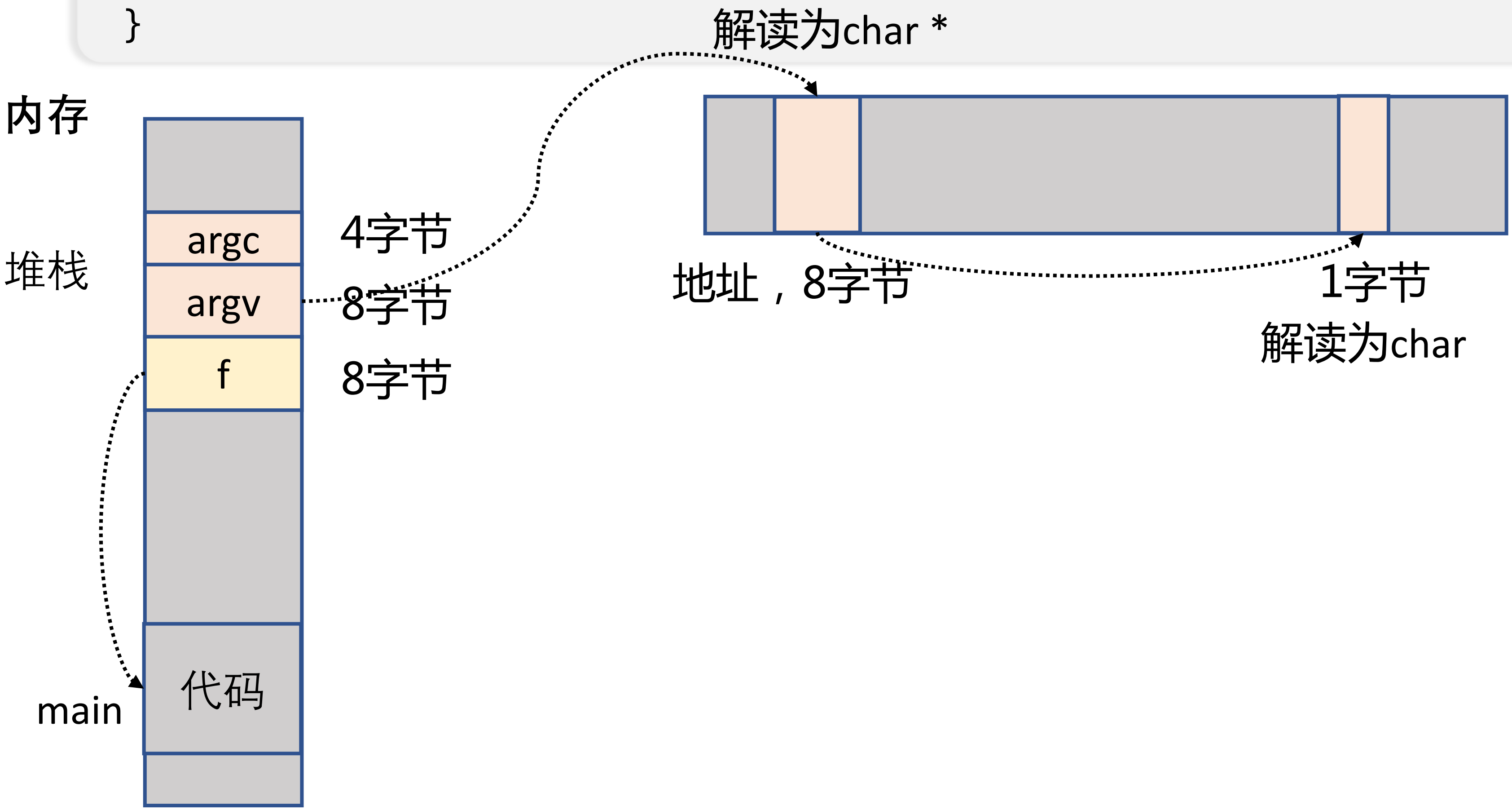
```
int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first
        printf("arg = \"%s\"; ch
        assert(***a == ch);
        f(argc - 1, argv + 1);
    }
}
```



char *argv[] ➔ char **argv ➔ (char *)*argv

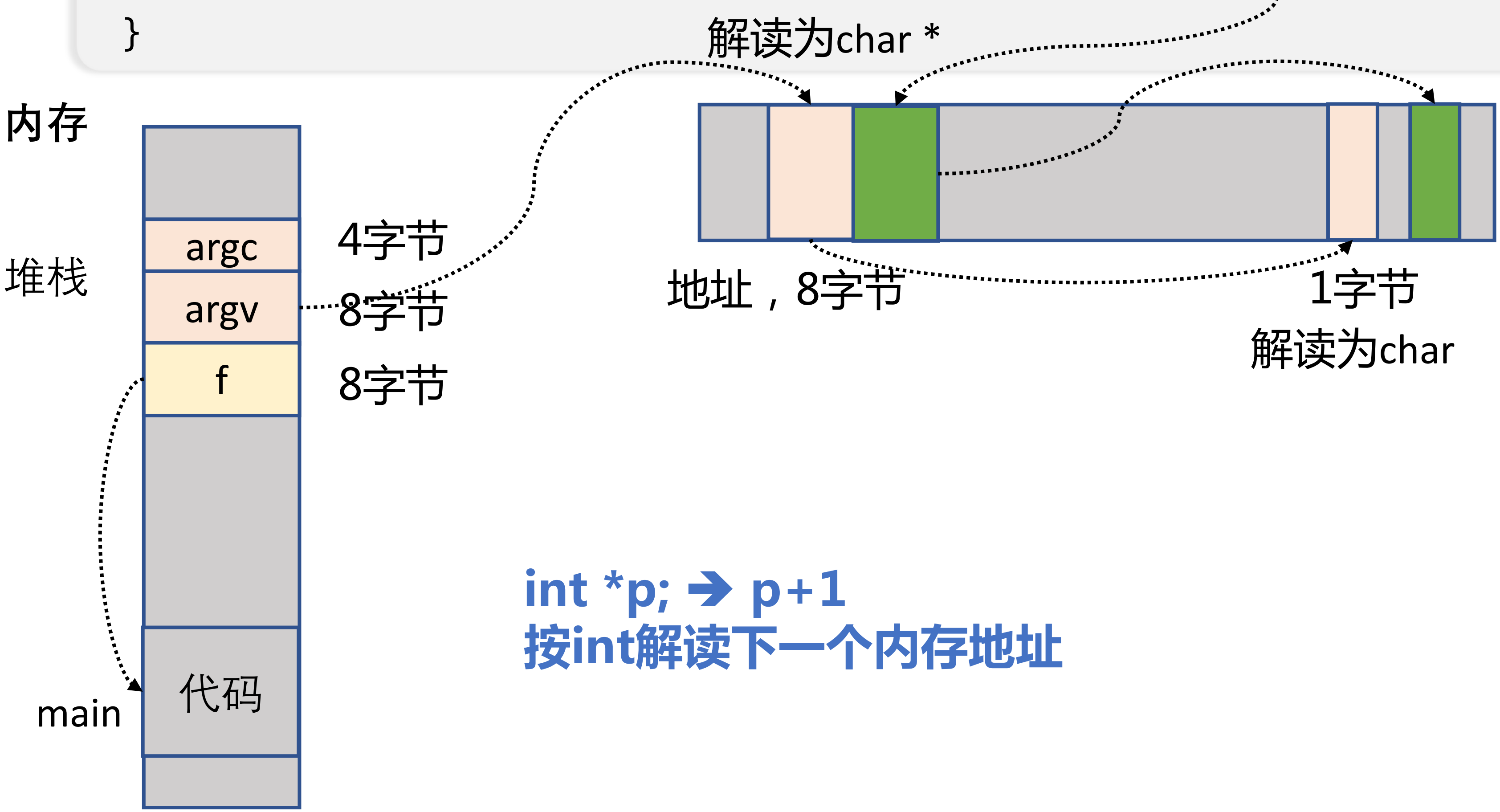


```c
int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(***a == ch);
        f(argc - 1, argv + 1);
    }
}
```



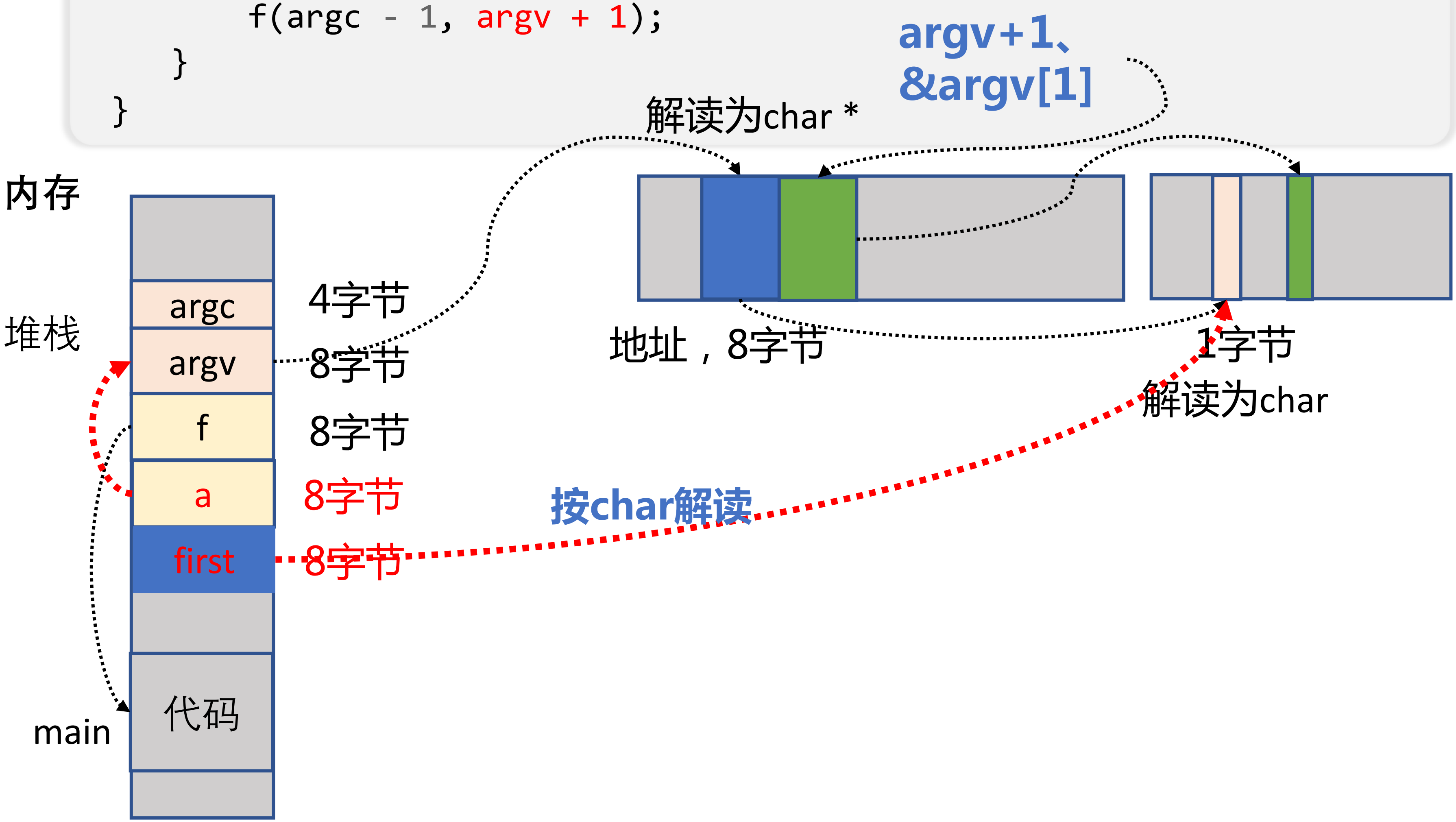
```c
int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(***a == ch);
        f(argc - 1, argv + 1);
    }
}
```

什么是argv+1？

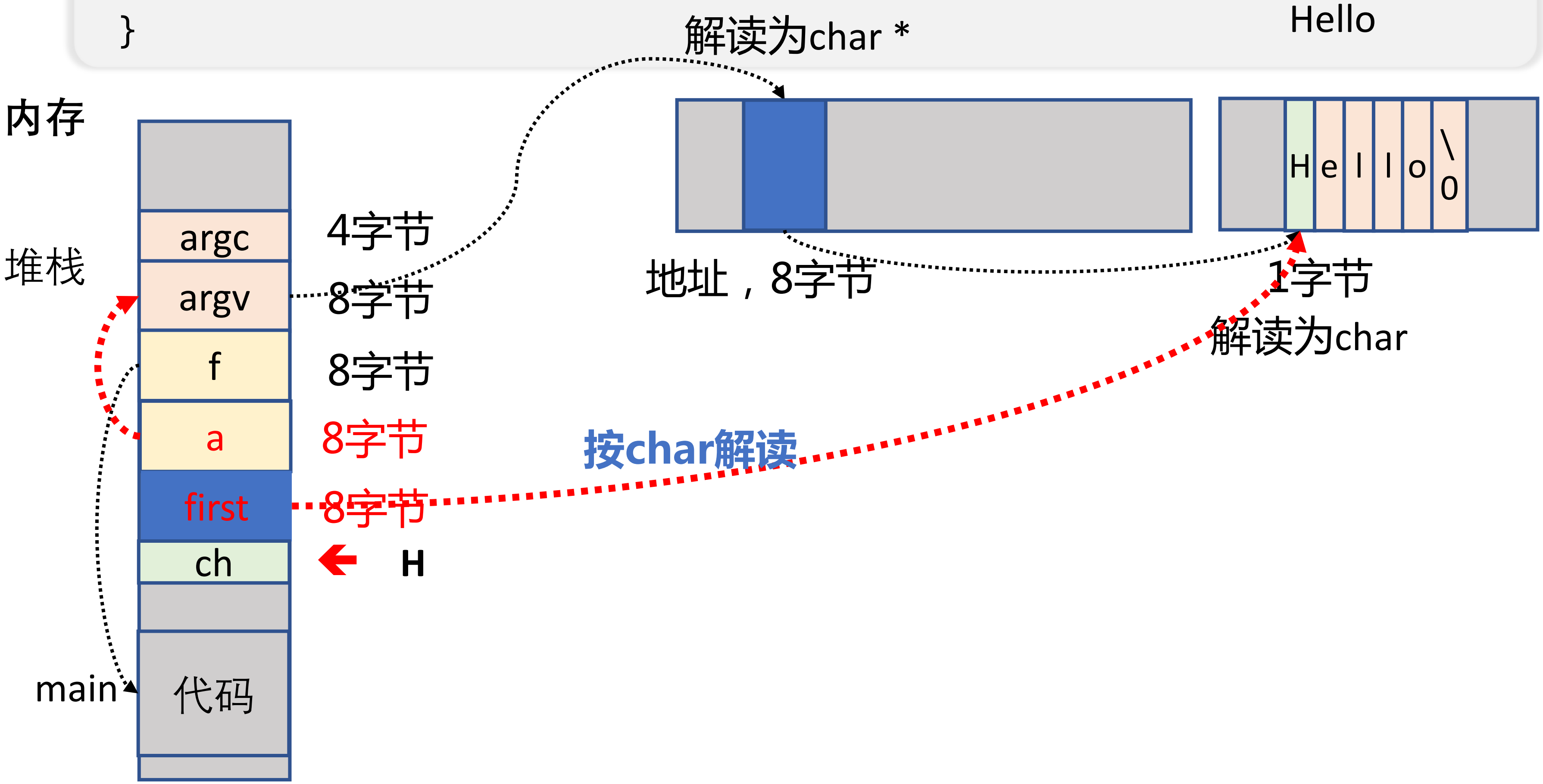


int *p; ➔ p+1
按int解读下一个内存地址

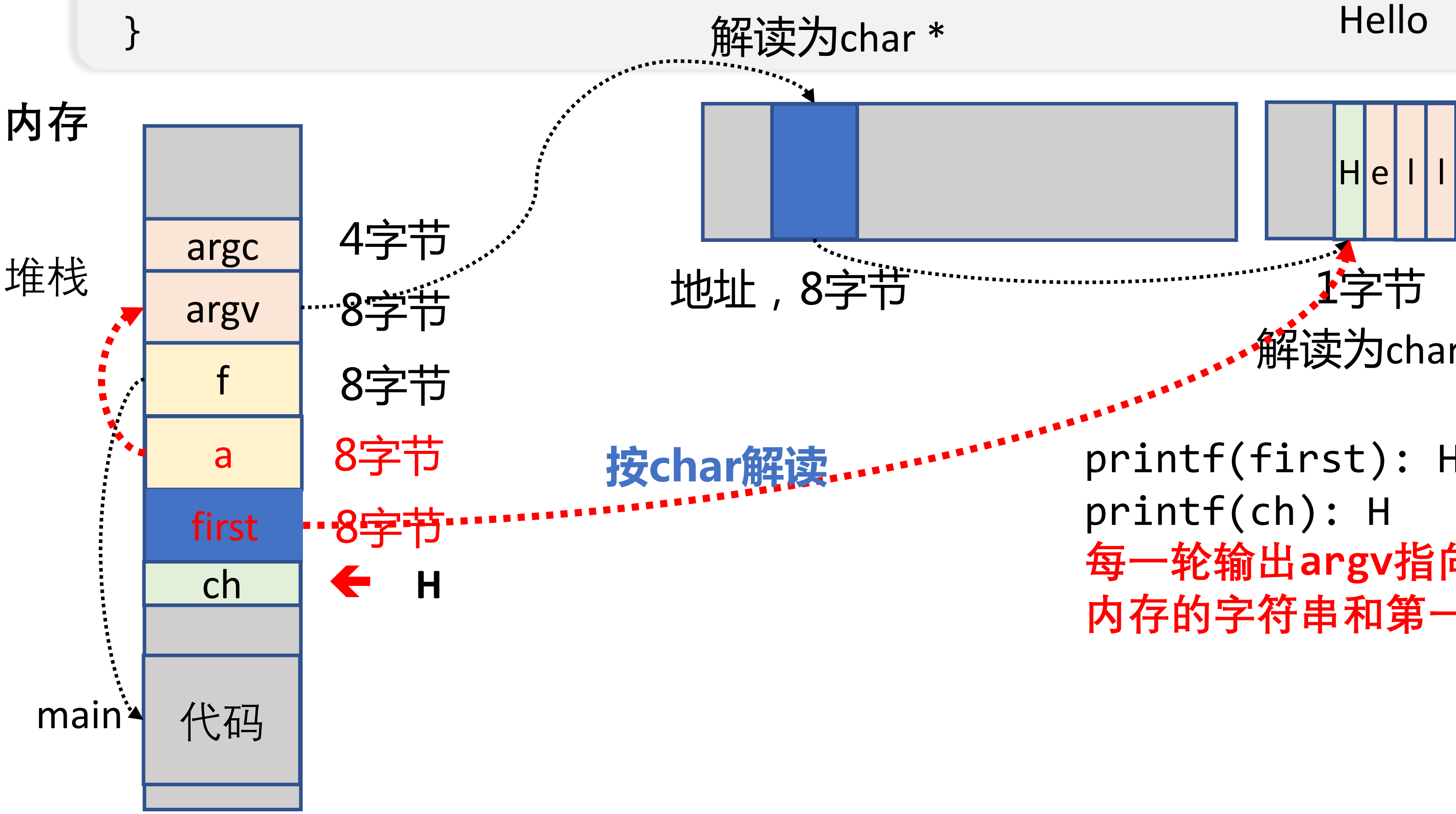
```c
int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(***a == ch);
        f(argc - 1, argv + 1);
    }
}
```



```c
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);
    }
}
```



```c
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);
    }
}
```



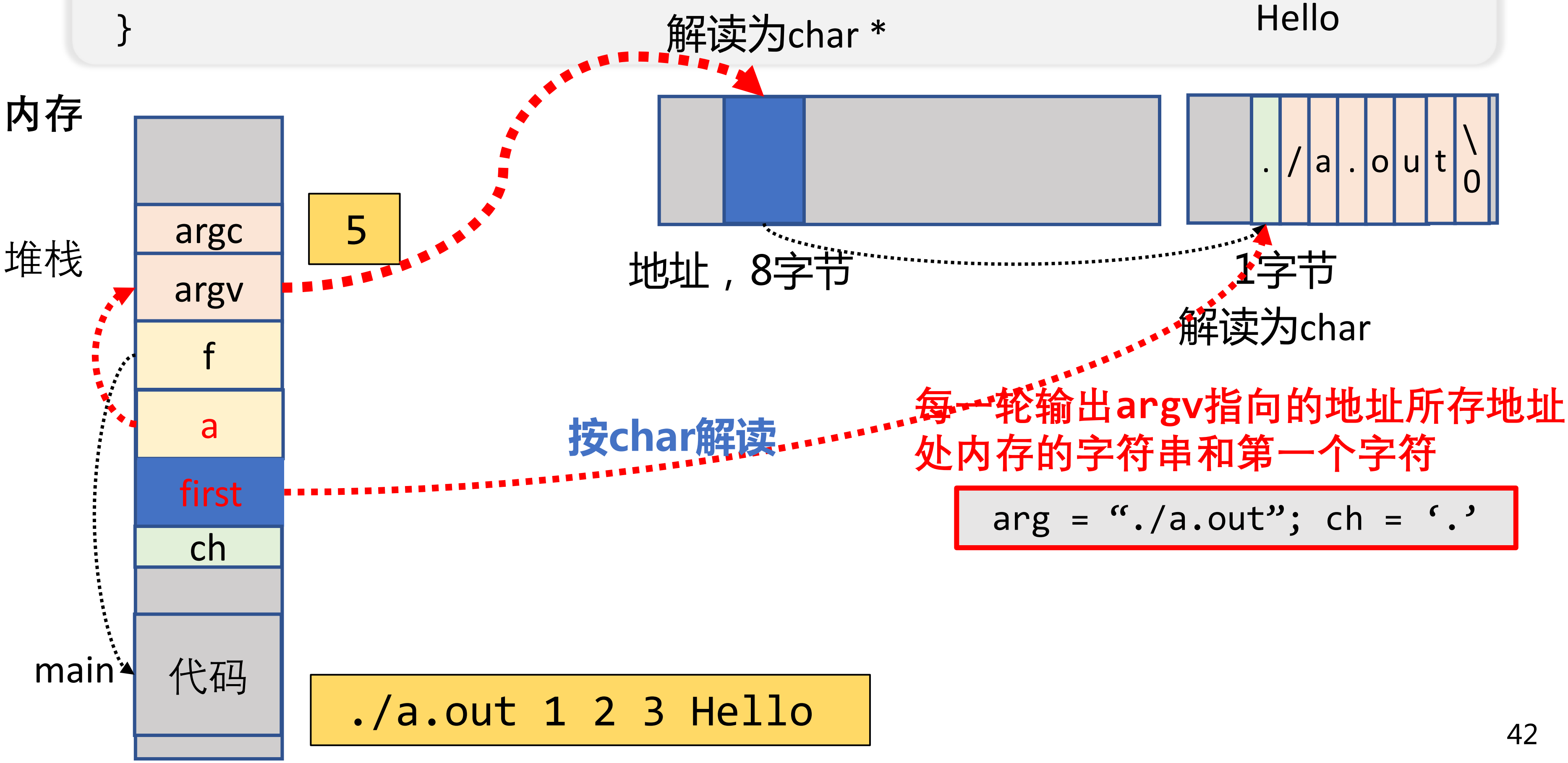
printf(first): Hello
printf(ch): H

每一轮输出argv指向的地址处内存的字符串和第一个字符

```c
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'
        assert(***a == ch);
        f(argc - 1, argv + 1);
    }
}
```

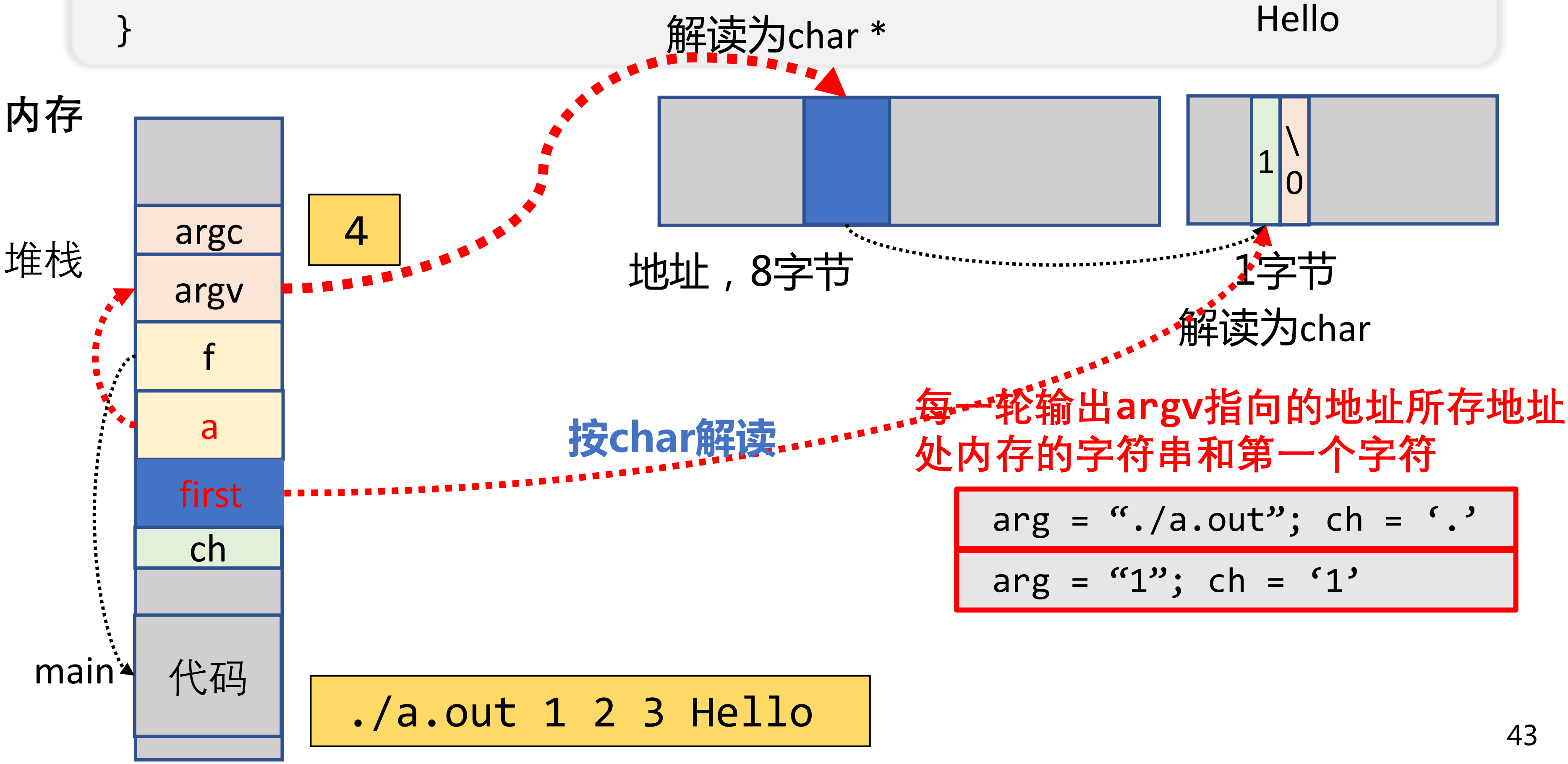
```
$ ./a.out 1 2 3 hello
arg = "./a.out";  ch = '.'
arg = "1";  ch = '1'
arg = "2";  ch = '2'
arg = "3";  ch = '3'
arg = "hello";  ch = 'h'
```

Hello



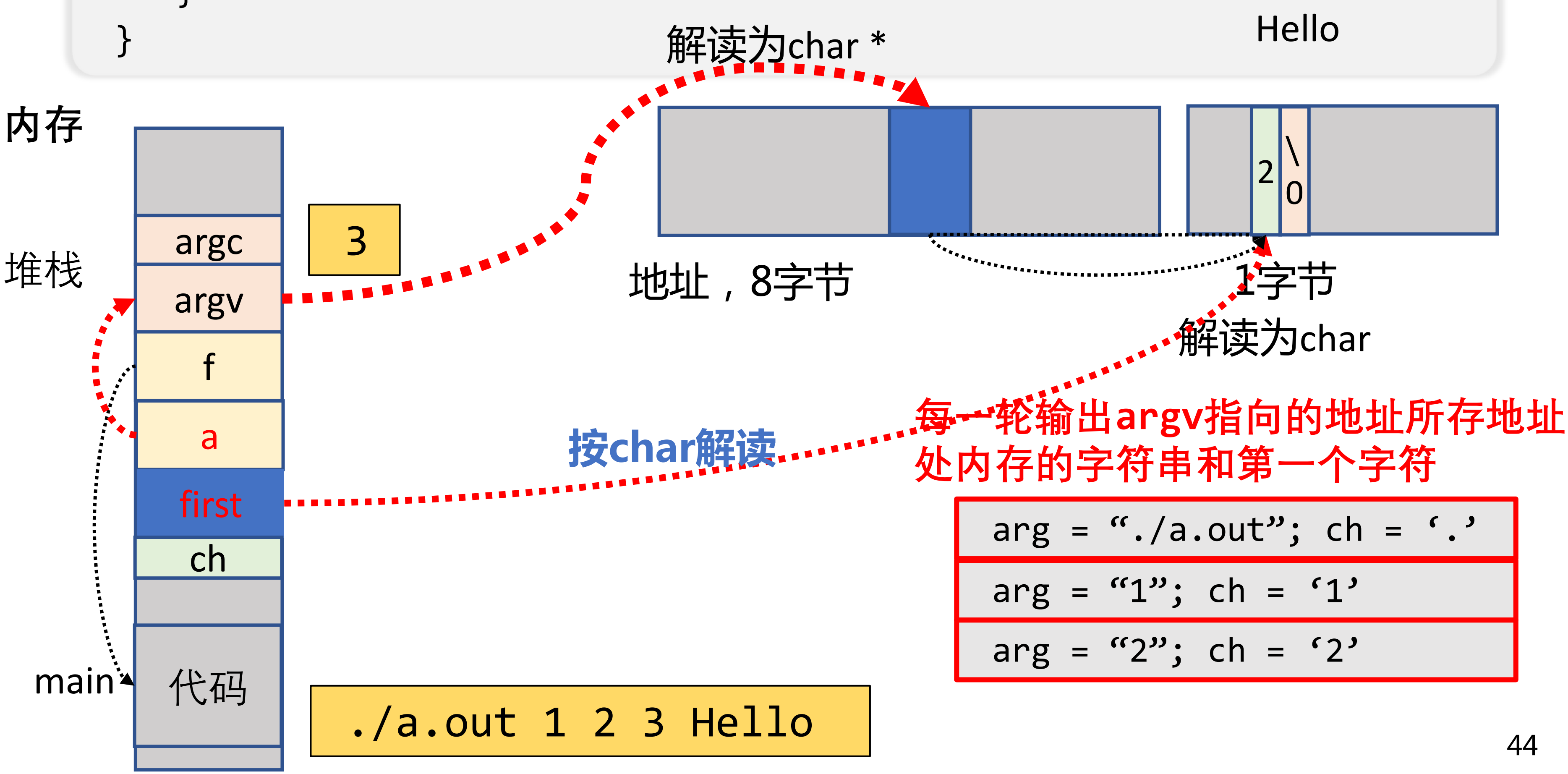
```c
int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0
        printf("arg = \"%s\"; ch = '%c'\n'
        assert(***a == ch);
        f(argc - 1, argv + 1);
    }
}
```

```
$ ./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'
```



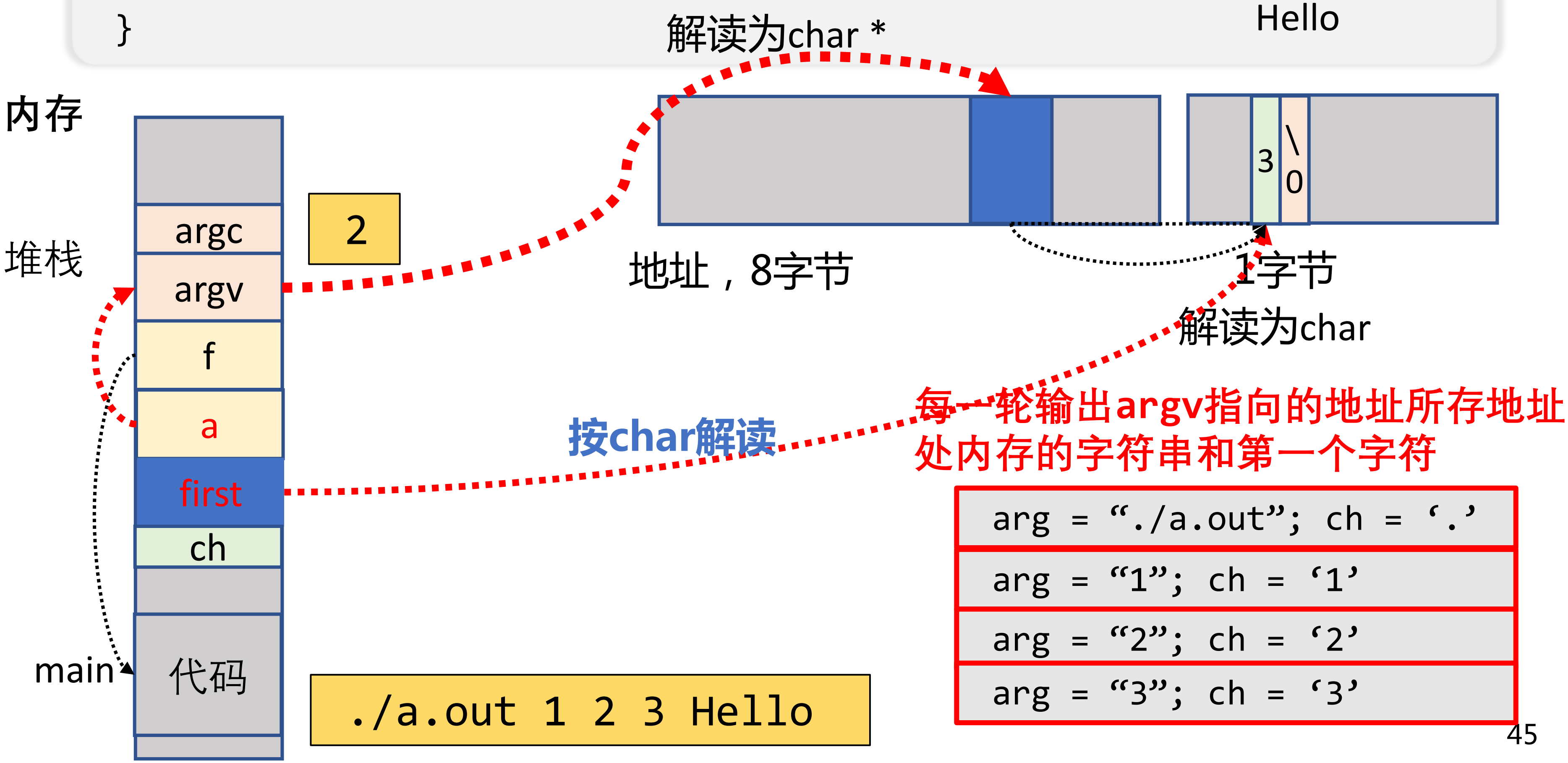
```
int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[(
        printf("arg = \"%s\"; ch = '%c'\n'
        assert(***a == ch);
        f(argc - 1, argv + 1);
    }
}
```

```
$ ./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'
```



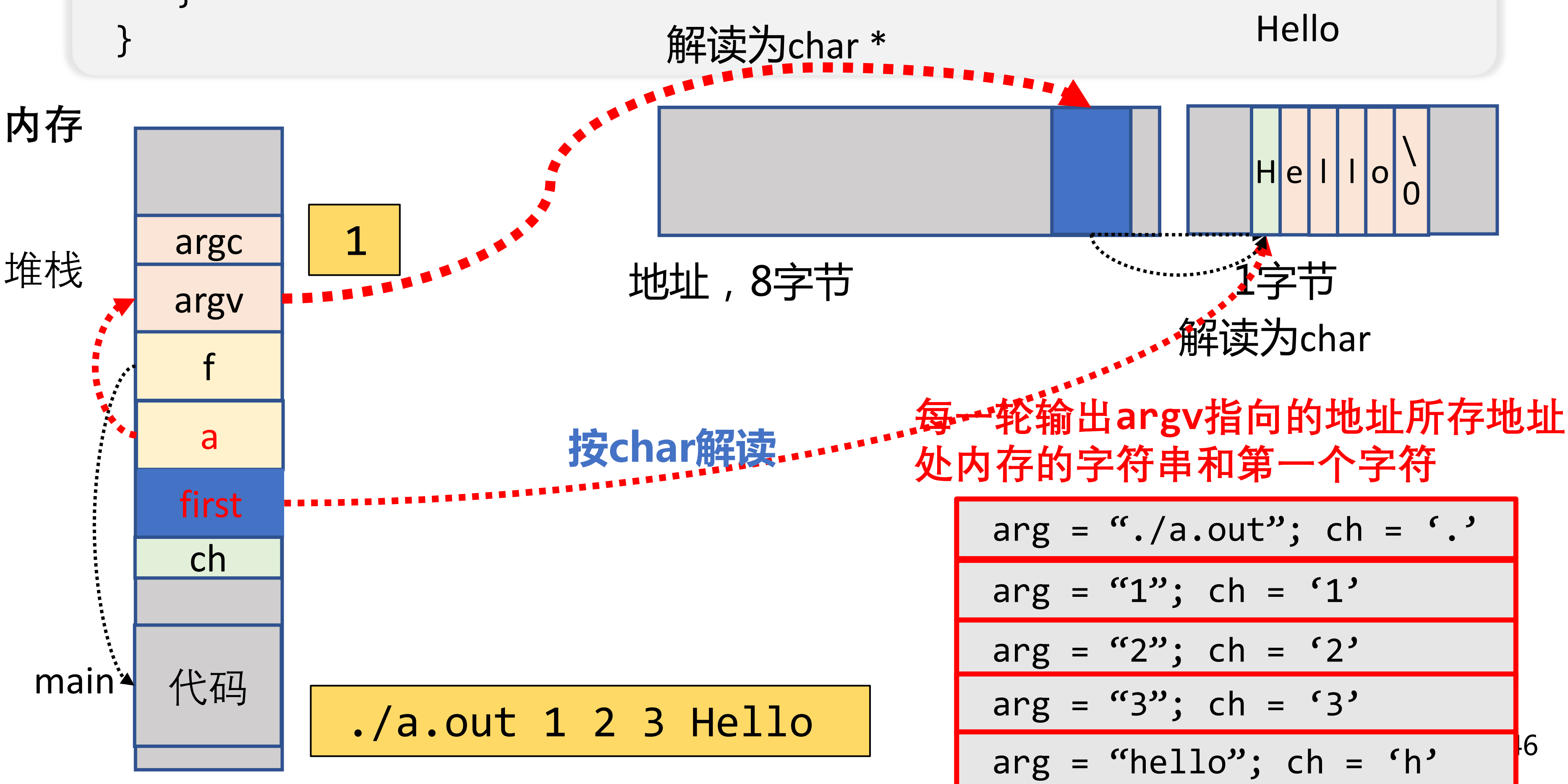
```c
int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0
        printf("arg = \"%s\"; ch = '%c'\n'
        assert(***a == ch);
        f(argc - 1, argv + 1);
    }
}
```

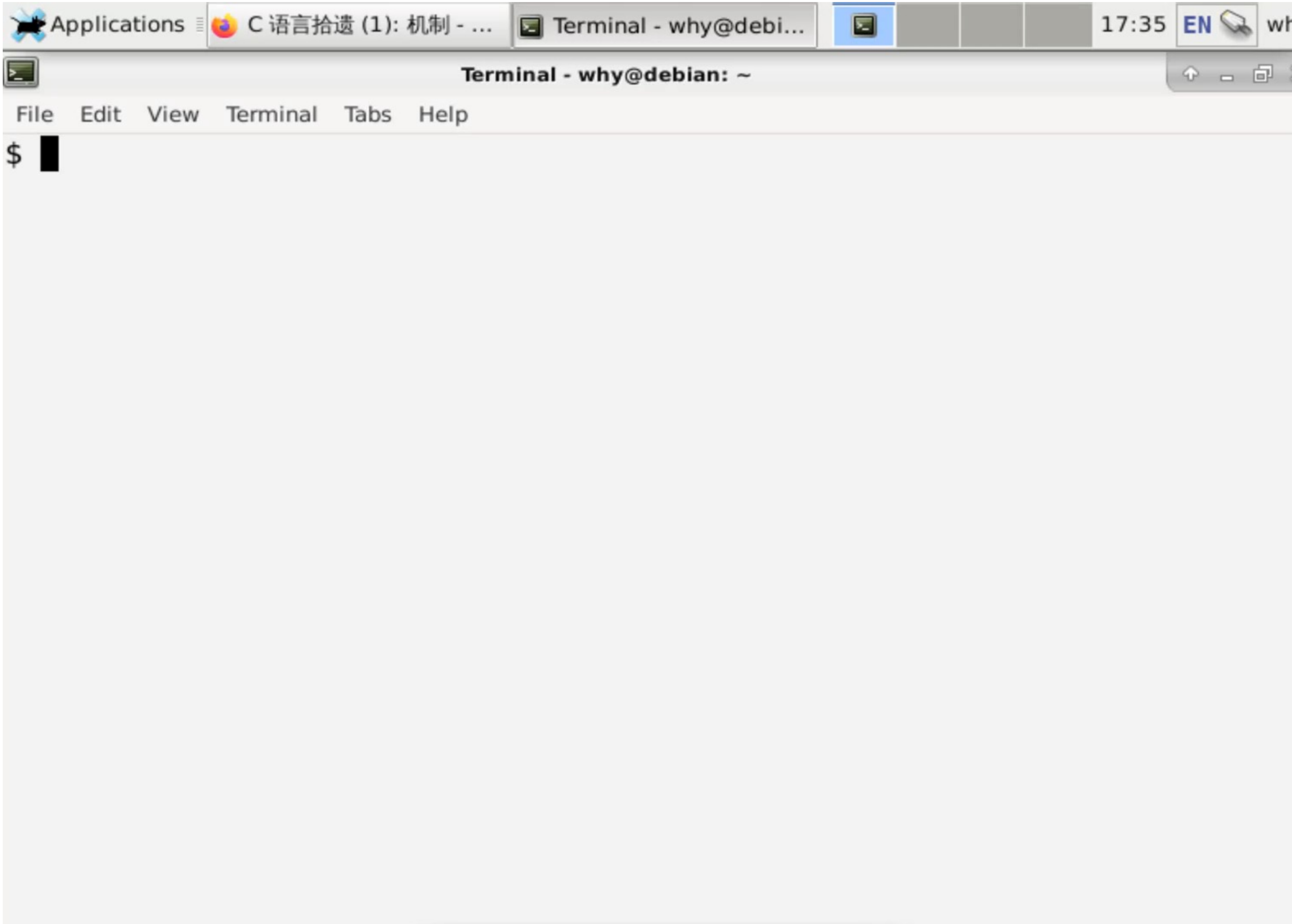
```
$ ./a.out 1 2 3 hello
arg = "./a.out";  ch = '.'
arg = "1";  ch = '1'
arg = "2";  ch = '2'
arg = "3";  ch = '3'
arg = "hello";  ch = 'h'
```



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




Applications
C 语言拾遗 (1): 机制 - ...
Terminal - why@debi...
17:35
EN
why
Terminal - why@debian: ~
File Edit View Terminal Tabs Help
$

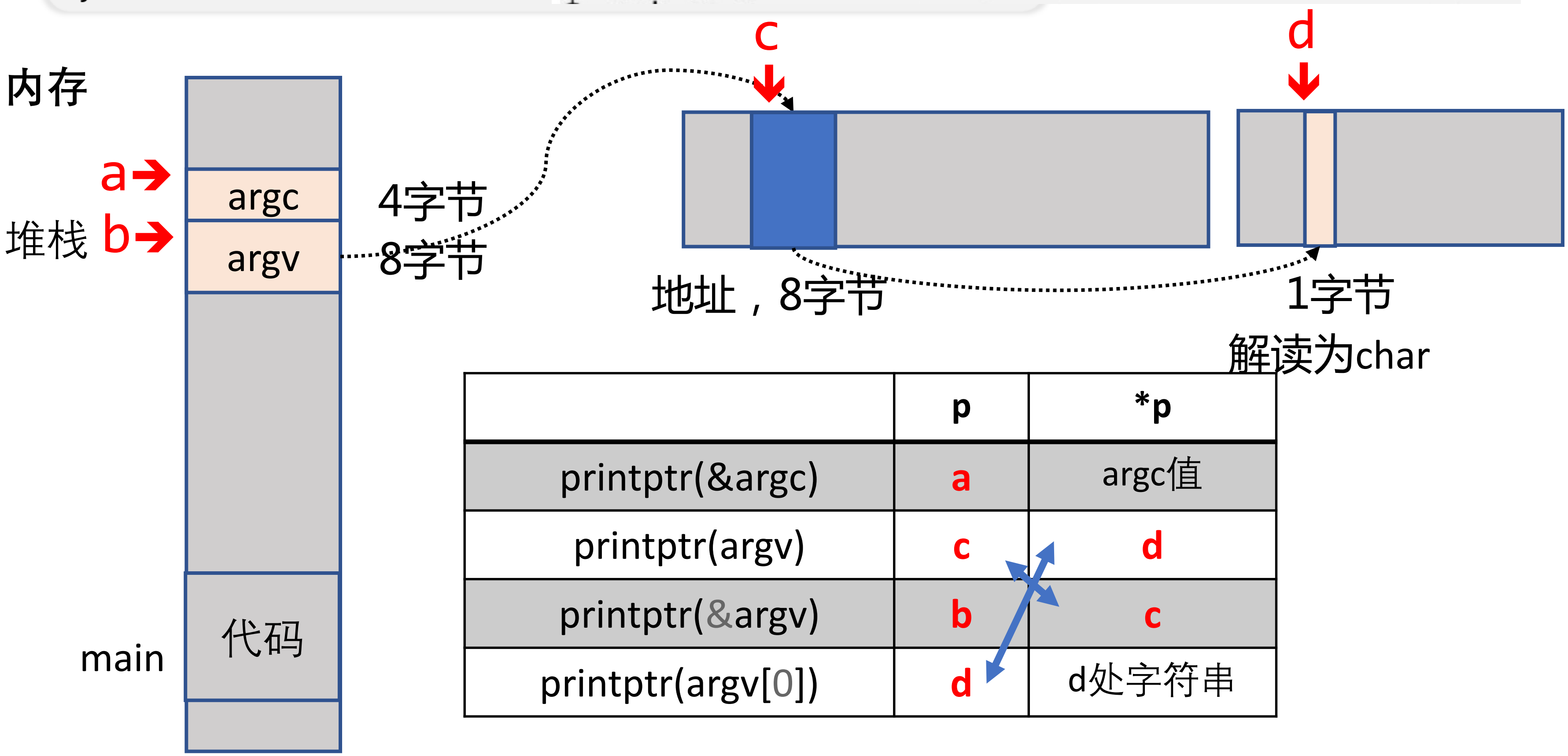
```c
void printptr(void *p) {
    printf("p = %p; *p = %016lx\n", p, *(long *)p);
}
int x;
int main(int argc, char *argv[]) {
    printptr(main); // 代码
    printptr(&main);
    printptr(&x); // 数据
    printptr(&argc); // 堆栈
    printptr(argv);
    printptr(&argv);
    printptr(argv[0]);
}
```

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

```
$ ./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



	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]`

核心准则：编写可读代码

怎样写代码才能从一个大型项目中存活下来？

核心准则：编写可读代码

一个极端不可读的例子

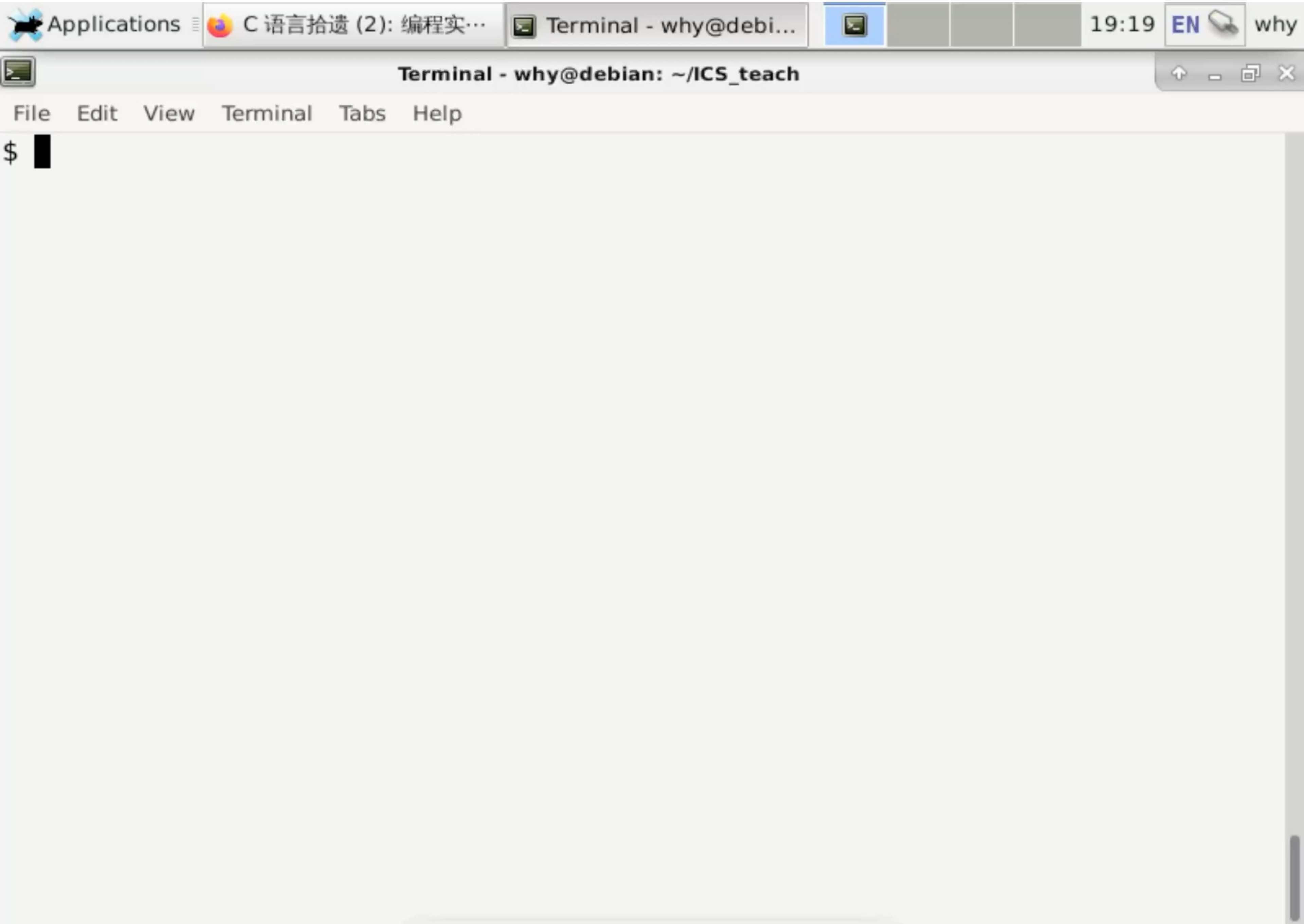
- [IOCCC'11 best self documenting program]
  - 不可读 = 不可维护

```
puts(usage: calculator 11/26+222/31
+~~~~~~~~~~~~~~~~~~~~~~~~calculator-\
!                       7.584,367 )
+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
! clear ! 0 ||l -x l tan I (/) |
+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
! 1 | 2 | 3 ||l 1/x l cos I (*) |
+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
! 4 | 5 | 6 ||l exp l sqrt I (+) |
+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
! 7 | 8 | 9 ||l sin l log I (-) |
+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~(0 )
```

一个极端不可读的例子

- IOCCC'11 best self documenting program
  - 不可读 = 不可维护

```
#define clear 1;
   if(c>=11){c=0;sscanf(_,"%lf%c",&r,&c);while(*++_-
c);}\     else if(argc>=4&&!main(4-
(*_++=='('),argv))_++;g:c+=
#define puts(d,e) return 0;}{double a;int b;char
   c=(argc<4?d)&15;\ b=(*_%__LINE__+7)%9*(3*e>>c&1);c+=
#define I(d)
(r);if(argc<4&&*#d==*_){a=r;r=usage?r*a:r+a;goto
   g;}c=c
#define return if(argc==2)printf("%f\n",r);return
argc>=4+ #define usage main(4-__LINE__/26,argv)
#define calculator *_*(int)
#define l (r);r=--b?r:
#define _ argv[1]
```


Applications
C 语言拾遗 (2): 编程实···
Terminal - why@debi...
19:19
EN
why
Terminal - why@debian: ~/ICS_teach
File Edit View Terminal Tabs Help
$

一个现实中可能遇到的例子

- 人类不可读版本（STFW: clockwise/spiral rule）
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```

一个现实中可能遇到的例子

- 人类不可读版本（STFW: clockwise/spiral rule）
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```

- signal是一个函数
  - 参数为……
  - 返回值为……

一个现实中可能遇到的例子

- 人类不可读版本（STFW: clockwise/spiral rule）
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```

- signal是一个函数
  - 参数为int和一个函数指针（参数为int，返回值为void）
  - 返回值为……

一个现实中可能遇到的例子

- 人类不可读版本（STFW: clockwise/spiral rule）
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```

- signal是一个函数
  - 参数为int和一个函数指针（参数为int，返回值为void）
  - 返回值为一个……指针

一个现实中可能遇到的例子

- 人类不可读版本（STFW: clockwise/spiral rule）
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```

- signal是一个函数
  - 参数为int和一个指向函数的指针（函数参数为int，返回值为void）
  - 返回值为一个指向函数的指针（参数为…，返回值为…）

一个现实中可能遇到的例子

- 人类不可读版本（STFW: clockwise/spiral rule）
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```

- signal是一个函数
  - 参数为int和一个指向函数的指针（函数参数为int，返回值为void）
  - 返回值为一个指向函数的指针（参数为int，返回值为…）

一个现实中可能遇到的例子

- 人类不可读版本（STFW: clockwise/spiral rule）
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```

  - signal是一个函数
    - 参数为int和一个指向函数的指针（函数参数为int，返回值为void）
    - 返回值为一个指向函数的指针（参数为int，返回值为void）

- 太复杂了>_<，有没有更简单的办法

一个现实中可能遇到的例子

- 人类不可读版本（STFW: clockwise/spiral rule）
  - 终极使用顺时针螺旋法则

```
void (*signal (int sig, void (*func)(int)))(int);
```

- signal是一个函数
  - 参数为int和一个指向函数的指针（函数参数为int，返回值为void）
  - 返回值为一个指向函数的指针（参数为int，返回值为void）

```
void (*signal (int sig, void (*func)(int)))(int);
```

```
void (*signal (int sig, func))(int);
```

```
void (*)(int);
```

一个现实中可能遇到的例子

- 人类不可读版本（STFW: clockwise/spiral rule）
  - 终极使用顺时针螺旋法则

```
void (*signal (int sig, void (*func)(int)))(int);
```

- 人类可读版本

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int, sighandler_t);
```

man 2 signal

Applications | Terminal - why@debian... 09:42 EN why

Terminal - why@debian: ~

File Edit View Terminal Tabs Help

```
SIGNAL(2)                  Linux Programmer's Manual                  SIGNAL(2)

NAME
       signal - ANSI C signal handling

SYNOPSIS
       #include <signal.h>

       typedef void (*sighandler_t)(int);

       sighandler_t signal(int signum, sighandler_t handler);

DESCRIPTION
       The behavior of signal() varies across UNIX versions, and has also varied histori-
       cally across different versions of Linux.  Avoid its  use:  use  sigaction(2)  in-
       stead.  See Portability below.

       signal()  sets  the  disposition  of the signal signum to handler, which is either
       SIG_IGN, SIG_DFL, or the address of a programmer-defined function (a "signal  han-
       dler").

       If  the  signal signum is delivered to the process, then one of the following hap-
       pens:

       *  If the disposition is set to SIG_IGN, then the signal is ignored.

       *  If the disposition is set to SIG_DFL, then the default action  associated  with
          the signal (see signal(7)) occurs.

       *  If  the  disposition is set to a function, then first either the disposition is
          reset to SIG_DFL, or the signal is blocked (see Portability  below),  and  then
 Manual page signal(2) line 1 (press h for help or q to quit)
```

编写代码的准则：降低维护成本

Programs are meant to be read by humans and only incidentally for computers to execute. — D. E. Knuth

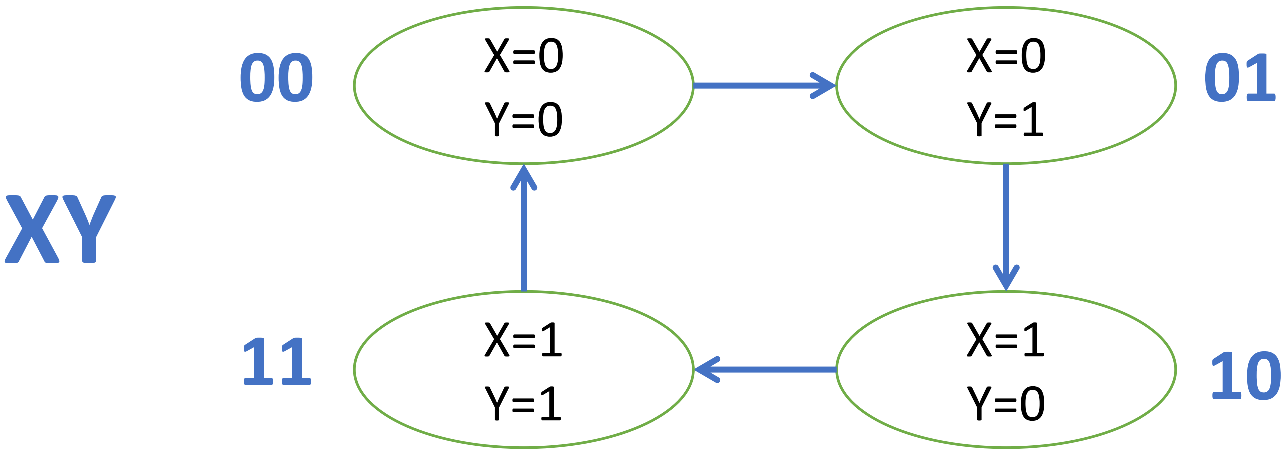
(程序首先是拿给人读的，其次才是被机器执行。)

- 宏观
  - 做好分解和解耦（现实世界也是这样管理复杂系统）
- 微观
  - “不言自明”
    - 通过阅读代码能够理解一段程序的行为（implementation）
  - “不言自证”
    - 通过阅读代码能够验证一段程序implementation与specification的一致性

例子：实现数字逻辑电路模拟器

数字逻辑电路模拟器

- 假想的数字逻辑电路
  - 若干个1-bit边沿触发寄存器（X, Y, …… ）
  - 若干个逻辑门



- 基本思路：状态（存储模拟）+计算模拟
  - 状态 = 变量
    - int X = 0, Y = 0;
  - 计算

```
X1 = (!X && Y) || (X && !Y);
Y1 = !Y;
X = X1; Y = Y1;
```

数字逻辑电路模拟器

- 需求
  - 加一位边沿寄存器
  - 自己独立的逻辑

```
int X=0, Y=0;
int  X1=0, Y1=0;
while(1){
        X1 = (!X&&Y)||(X&&!Y);
        Y1 = !Y;
        X = X1; Y = Y1;
}
```

```
int X=0, Y=0, Z=0;
int  X1=0, Y1=0, Z1=0;
while(1){
        X1 = (!X&&Y)||(X&&!Y);
        Y1 = !Y;
        Z1 = ……;
        X = X1; Y = Y1; Z = Z1;
}
```

通用数字逻辑电路模拟器

```c
#define FORALL_REGS(_)   _(X) _(Y)
#define LOGIC            X1 = (!X&&Y)||(X&&!Y); \
                         Y1 = !Y;
#define DEFINE(X)        static int X, X##1;
#define UPDATE(X)        X = X##1;
#define PRINT(X)         printf(#X " = %d; ", X);
int main() {
  FORALL_REGS(DEFINE);

  while (1) { // clock
    FORALL_REGS(PRINT);
    putchar('\n’);
    sleep(1);
    LOGIC;
    FORALL_REGS(UPDATE);
  }
}
```

Applications | C 语言拾遗 (2): 编程实… | Terminal - a.c (~/ICS_… | 11:34 EN why

Terminal - a.c (~/ICS_teach) - VIM

File Edit View Terminal Tabs Help

```c
#include <stdio.h>
#include <unistd.h>
#define FORALL_REGS(_)  _(X) _(Y)
#define LOGIC           X1 = (!X && Y)||(X&&!Y); \
                        Y1 = !Y;
#define DEFINE(X)       static int X, X##1;
#define UPDATE(X)       X = X##1;
#define PRINT(X)        printf(#X " = %d; ", X);

int main() {
  FORALL_REGS(DEFINE);
  while (1) { // clock
    FORALL_REGS(PRINT); putchar('\n'); sleep(1);
    LOGIC;
    FORALL_REGS(UPDATE);
  }
}

```

```
a.c                                                   3,1            All
"a.c" 18L, 437C
```

Applications | C 语言拾遗 (2): 编程实… | Terminal - a.c (~/ICS_… | 11:54 EN why

Terminal - a.c (~/ICS_teach) - VIM

File Edit View Terminal Tabs Help

```c
#include <stdio.h>
#include <unistd.h>
#define FORALL_REGS(_)  _(X) _(Y) _(Z)
#define LOGIC           X1 = (!X&&Y&&Z)||(X&&(!(Y&&Z))); \
                        Y1 = (!Y&&Z)||(Y&&!Z); \
                        Z1 = !Z;
#define DEFINE(X)       static int X, X##1;
#define UPDATE(X)       X = X##1;
#define PRINT(X)        printf(#X " = %d; ", X);

int main() {
  FORALL_REGS(DEFINE);
  while (1) { // clock
    FORALL_REGS(PRINT); putchar('\n'); sleep(1);
    LOGIC;
    FORALL_REGS(UPDATE);
  }
}

a.c                                                   4,1          All
"a.c" 19L, 498C
```

通用数字逻辑电路模拟器

```c
#define FORALL_REGS(_)   _(X) _(Y)
#define LOGIC            X1 = (!X&&Y)||(X&&!Y); \
                         Y1 = !Y;
#define DEFINE(X)        static int X, X##1;
#define UPDATE(X)        X = X##1;
#define PRINT(X)         printf(#X " = %d; ", X);
int main() {
   FORALL_REGS(DEFINE);

   while (1) { // clock
       FORALL_REGS(PRINT);
       putchar('\n’);
       sleep(1);
       LOGIC;
       FORALL_REGS(UPDATE);
   }
}
```

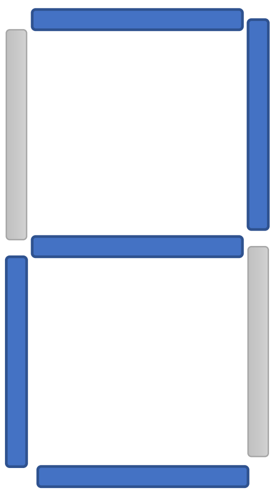
使用预编译：Pros and Cons

- Pros
  - 增加/删除寄存器只要改很少的地方
  - 阻止一些编程错误
    - 忘记更新寄存器
    - 忘记打印寄存器
  - “不言自明”

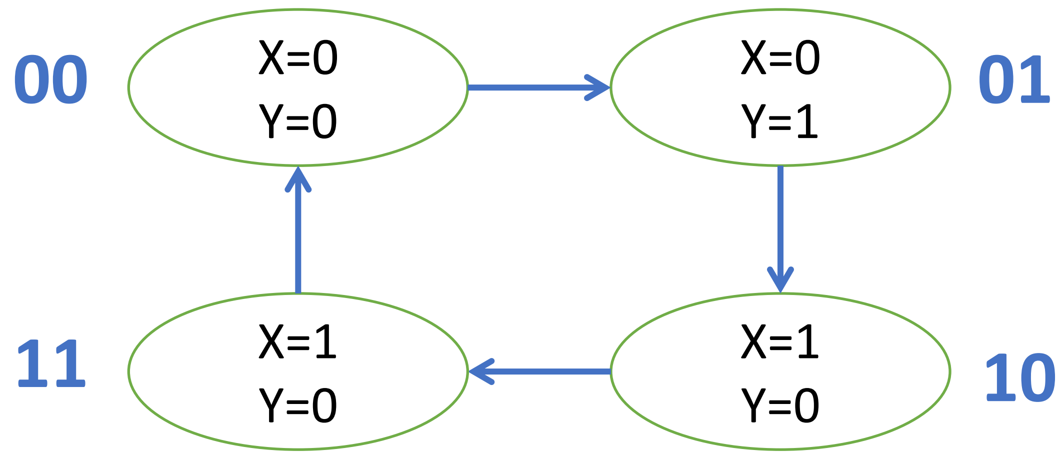
- Cons
  - 可读性变差（不太像C代码）
    - “不言自证”差一些
  - 给IDE解析带来一些困难

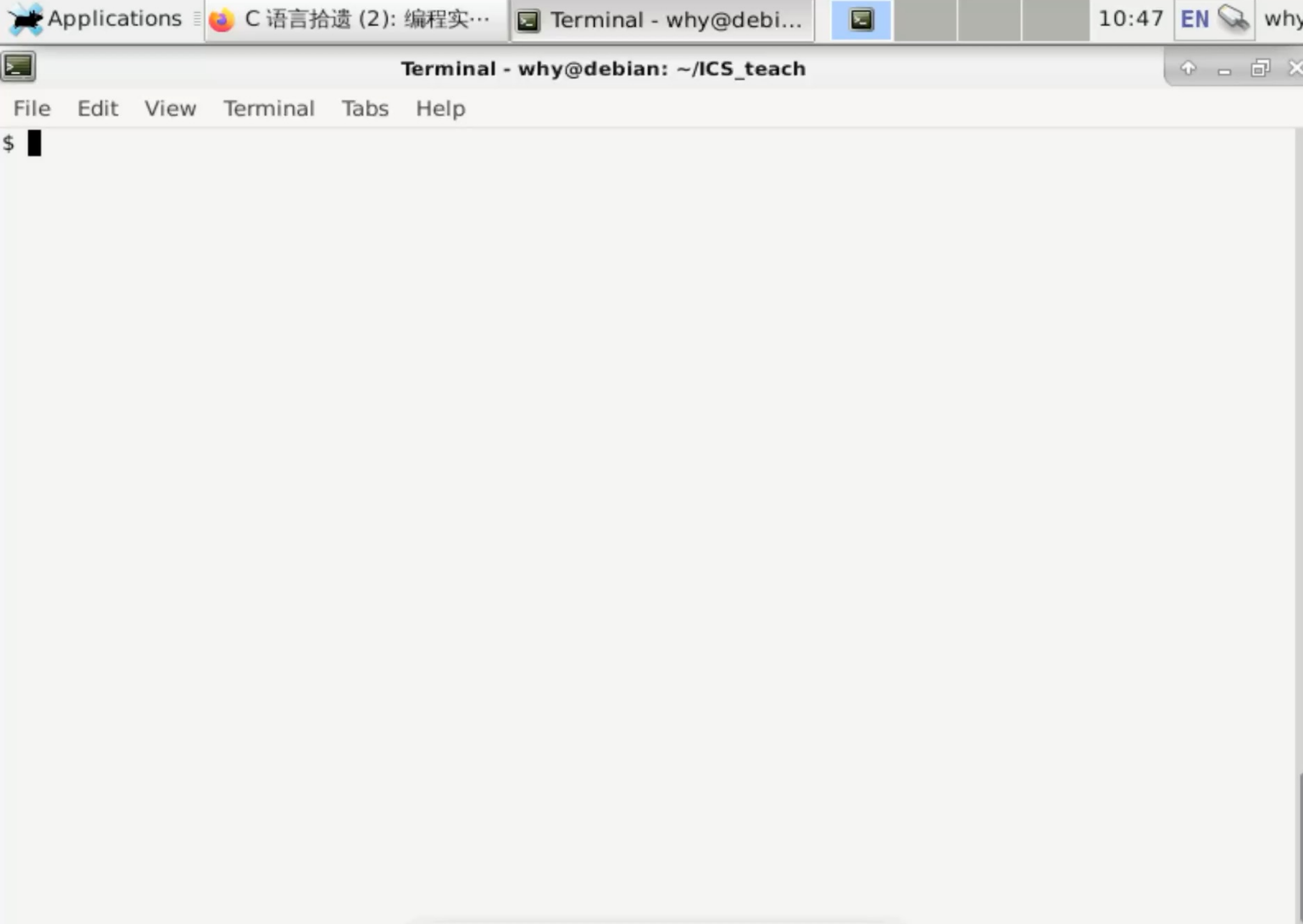
更完整的实现：数码管显示

- logisim.c 和 display.py



XY




Applications
C 语言拾遗 (2): 编程实⋯
Terminal - why@debi...
10:47
EN
why
Terminal - why@debian: ~/ICS_teach
File Edit View Terminal Tabs Help
$

```c
#include <stdio.h>
#include <unistd.h>
#define FORALL_REGS(_)      _(X) _(Y)
#define OUTPUTS(_)          _(A) _(B) _(C) _(D) _(E) _(F) _(G)
#define LOGIC               X1 = (!X && Y) || (X && !Y); \
                            Y1 = !Y; \
                            A  = (!X && !Y) || (X && !Y) || (X && Y); \
                            B  = 1; \
                            C  = (!X && !Y) || (!X && Y) || (X && Y); \
                            D  = (!X && !Y) || (X && !Y) || (X && Y); \
                            E  = (!X && !Y) || (X && !Y); \
                            F  = (!X && !Y); \
                            G  = (X && !Y) || (X && Y);
#define DEFINE(X)           static int X, X##1;
#define UPDATE(X)           X = X##1;
#define PRINT(X)            printf(#X " = %d; ", X);
int main() {
    FORALL_REGS(DEFINE);
    OUTPUTS(DEFINE);
    while (1) {// clock
        LOGIC;
        OUTPUTS(PRINT);
        putchar('\n');
        fflush(stdout);
        sleep(1);
        FORALL_REGS(UPDATE);
    }
}
```

更完整的实现：数码管显示

- logisim.c 和 display.py
  - 也可以考虑增加更多外设：开关、UART等
  - 原理无限接近数字电路课玩过的FPGA

- 等等……FPGA？
  - 这玩意不是万能的吗？？？
  - 我们能模拟它，是不是就能模拟计算机系统？
    - Yes!
    - 我们实现了一个超级超级低配版 NEMU!

例子：实现YEMU全系统模拟器

更多的计算机系统模拟器

- `am-kernels/litenes`
  - 一个 “最小” 的 NES 模拟器
  - 自带硬编码的 ROM 文件
- `fceux-am`
  - 一个非常完整的高性能 NES 模拟器
  - 包含对卡带定制芯片的模拟 (`src/boards`)
- QEMU
  - 工业级的全系统模拟器
    - 2011 年发布 1.0 版本
    - 有兴趣的同学可以 RTFSC
  - 作者：传奇黑客 Fabrice Bellard

永远不要停止对好代码的追求

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