

# Pointer++

王慧妍

[why@nju.edu.cn](mailto:why@nju.edu.cn)

南京大学



软件学院



计算机软件研究所



# String.h

- strcmp/strncmp
  - [strcmp.c](#)
  - [strcmp - cppreference.com](#)
- strcat
  - [concat.c](#)
- memcmp
  - [memcmp.c](#)

---

## THE STANDARD

<time.h> \* <limits.h> \* <float.h>  
<stddef.h> \* <ctype.h> \* <locale.h>  
<stdio.h> \* <string.h>  
<math.h> \* <stdlib.h> \* <unistd.h>  
<stdarg.h> \* <setjmp.h> \* <signal.h>  
<time.h> \* <limits.h> \* <float.h>  
<stddef.h> \* <errno.h> \* <locale.h>  
<stdio.h> \* <ctype.h> \* <string.h>  
<math.h> \* <stdlib.h> \* <unistd.h>  
<stdarg.h> \* <setjmp.h> \* <signal.h>  
<time.h> \* <limits.h> \* <float.h>  
<stddef.h> \* <errno.h> \* <locale.h>

## LIBRARY

P.J. PLAUCER

---

# 比较指针数组和数组指针

---

- `char * array[N];`
- `char (* array)[N];`
- `char array[N][M];`

# 字符串回顾

---

- `char * array[N];`
- selectsort排序字符串数组
  - strcmp
    - `int __cdecl strcmp(const char *_Str1,const char *_Str2);`
  - 思考：每一轮排序`swap`的是什么？
  - [pointer-array-sort.c](#)

# 指针数组的另外用途

- 传递命令行参数
  - [echo.c](#)
  - `./echo hello world`

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

- `./echo -a hello -b world -c`
  - `getopt/getopt_long`

# 高级指针

---

- 这又是什么意思？
  - `int (*f[])();`
  - `int *(*f[])();`
  - `int (*f)(int, double);`
  - `int *(*g[])(int, double);`

# 函数指针

- 指针表示对某个特定内存地址的指向

```
return_type (*pointer_name)(parameter_types);
```

- 内存地址不仅能够表示变量
- 还可能代表某个函数的实现
  - 函数占用内存单元，每一个函数都有自己的地址
  - [point.c](#)

# 在C语言课上回顾点数学

- [Integration.c](#)

$$\int_a^b f(x)dx = \sum_{i=0}^{n-1} f(x_i) \left( \frac{b-a}{n} \cdot i \right)$$

$$a = x_0 < x_1 < \cdots x_i < \cdots < x_{n-1} < x_n = b$$

$$x_i = a + \frac{b-a}{n} \cdot i$$

- WolframAlpha (get the right answer!)
  - [integrate sin x dx from x=0 to 1 - Wolfram|Alpha \(wolframalpha.com\)](#)
  - [integrate cos x dx from x=0 to 1 - Wolfram|Alpha \(wolframalpha.com\)](#)



# 排序

---

- [sort.c](#)
- [qsort, qsort\\_s - cppreference.com](#)

# 函数指针

---

- `int (*comp) (const void *left, const void *right);`
- `int atexit(void (*func)(void));`
- `char (*(*arr[3])())[5];`
- `void (*signal (int sig, void (*func)(int)))(int);`
- `char (*(*func(int num, char*str))[])( );`

`char (*(arr[3])())[5];` 是一个复杂的声明，表示 `arr` 是一个包含 3 个元素的数组。数组中的每个元素都是一个指向函数的指针，这些函数返回一个指向包含 5 个 `char` 元素的数组的指针。

为了更好地理解这个声明，我们可以将其分解：

1. `arr[3]`： `arr` 是一个包含 3 个元素的数组。
2. `(*arr[3])()`： 数组中的每个元素是一个指向函数的指针。
3. `char (*(arr[3])())[5]`： 这些函数返回一个指向包含 5 个 `char` 元素的数组的指针。

`char (*(func(int num, char* str))[])( );` 是一个复杂的声明，表示 `func` 是一个函数，该函数接受两个参数：一个 `int` 类型和一个 `char*` 类型，并返回一个指向数组的指针，该数组的元素是指向返回 `char` 类型的函数的指针。

为了更好地理解这个声明，我们可以将其分解：

1. `func(int num, char* str)`： `func` 是一个函数，接受两个参数：一个 `int` 类型和一个 `char*` 类型。
2. `(*func(int num, char* str))`： `func` 返回一个指针。
3. `(*func(int num, char* str))[]`： 该指针指向一个数组。
4. `(*func(int num, char* str))[]()`： 数组的元素是指向函数的指针。
5. `char (*(func(int num, char* str))[])( )`： 这些函数返回 `char` 类型。

`void (*signal (int sig, void (*func)(int)))(int);` 是一个复杂的声明，表示 `signal` 是一个函数，该函数接受两个参数：

1. `int sig`： 一个整数类型的信号编号。
2. `void (*func)(int)`： 一个指向接受一个 `int` 参数并返回 `void` 的函数的指针。

`signal` 函数返回一个指向接受一个 `int` 参数并返回 `void` 的函数的指针。

# 一个现实中可能遇到的地狱难度例子

- 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
    - [Clockwise/Spiral Rule](#)

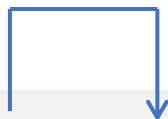
```
void (*signal (int sig, void (*func)(int)))(int);
```

# 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

- [Clockwise/Spiral Rule](#)



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

- signal是一个函数

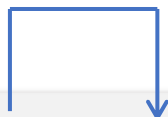
- 参数为.....
  - 返回值为.....

# 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

- [Clockwise/Spiral Rule](#)



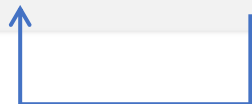
```
void (*signal (int sig, void (*func)(int)))(int);
```

- 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
    - [Clockwise/Spiral Rule](#)

```
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

- [Clockwise/Spiral Rule](#)



```
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

- [Clockwise/Spiral Rule](#)



```
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

- [Clockwise/Spiral Rule](#)

```
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

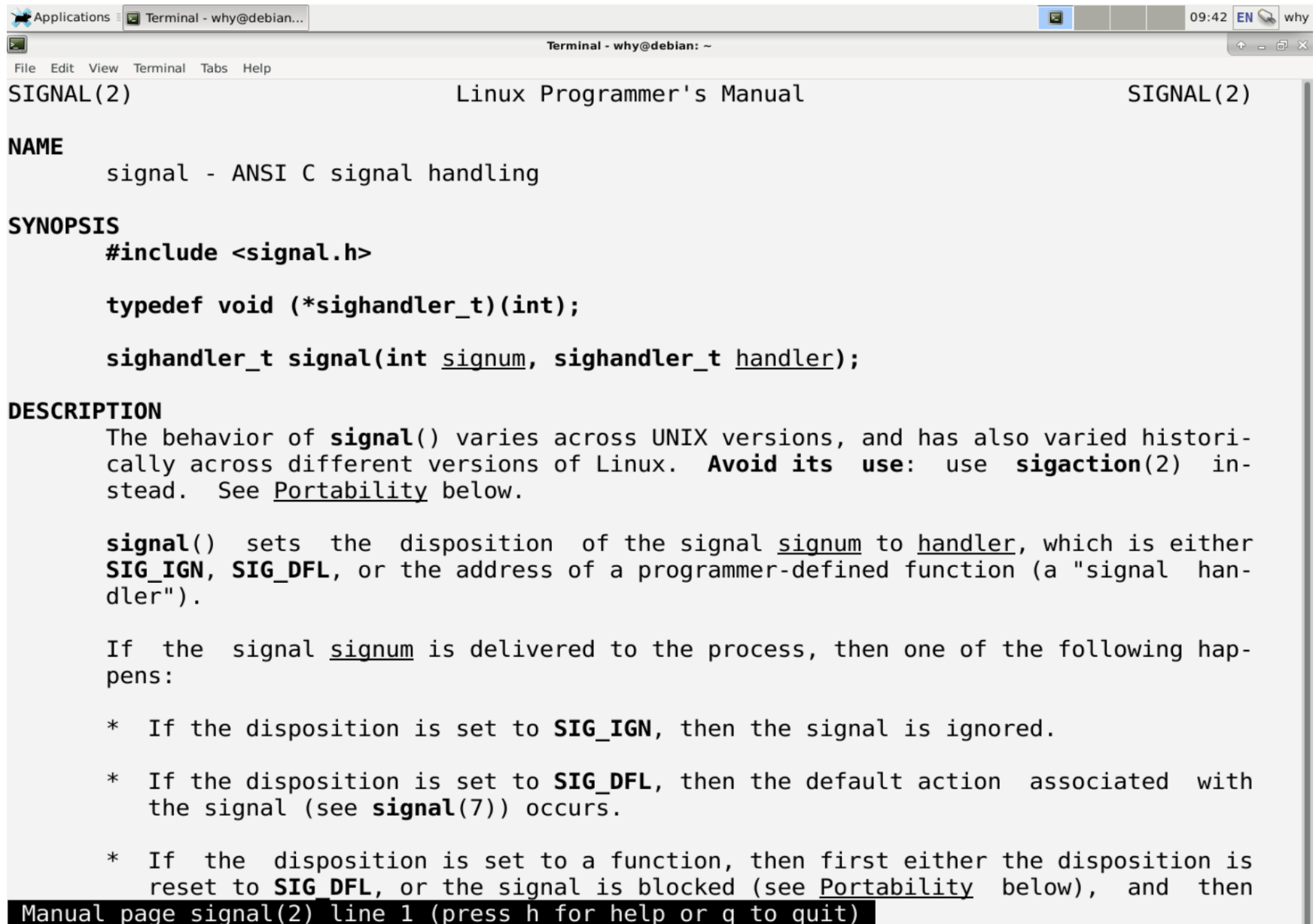
- 人类不可读版本 ( STFW: clockwise/spiral rule )
  - 终极使用顺时针螺旋法则

```
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 historically across different versions of Linux. Avoid its use: use sigaction(2) instead. 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 handler").

    If the signal signum is delivered to the process, then one of the following happens:

    * 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)
```

# End

---

- [cdecl: C gibberish ↔ English](#)

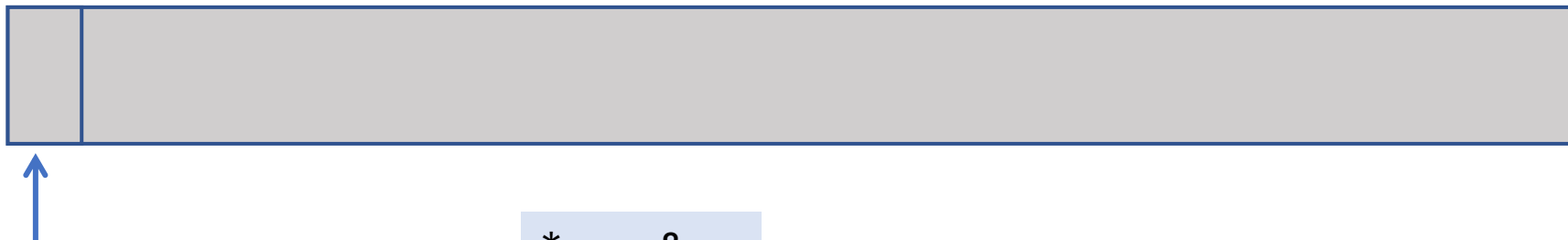
福利（？）：进入C语言的世界

课上没讲

# 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



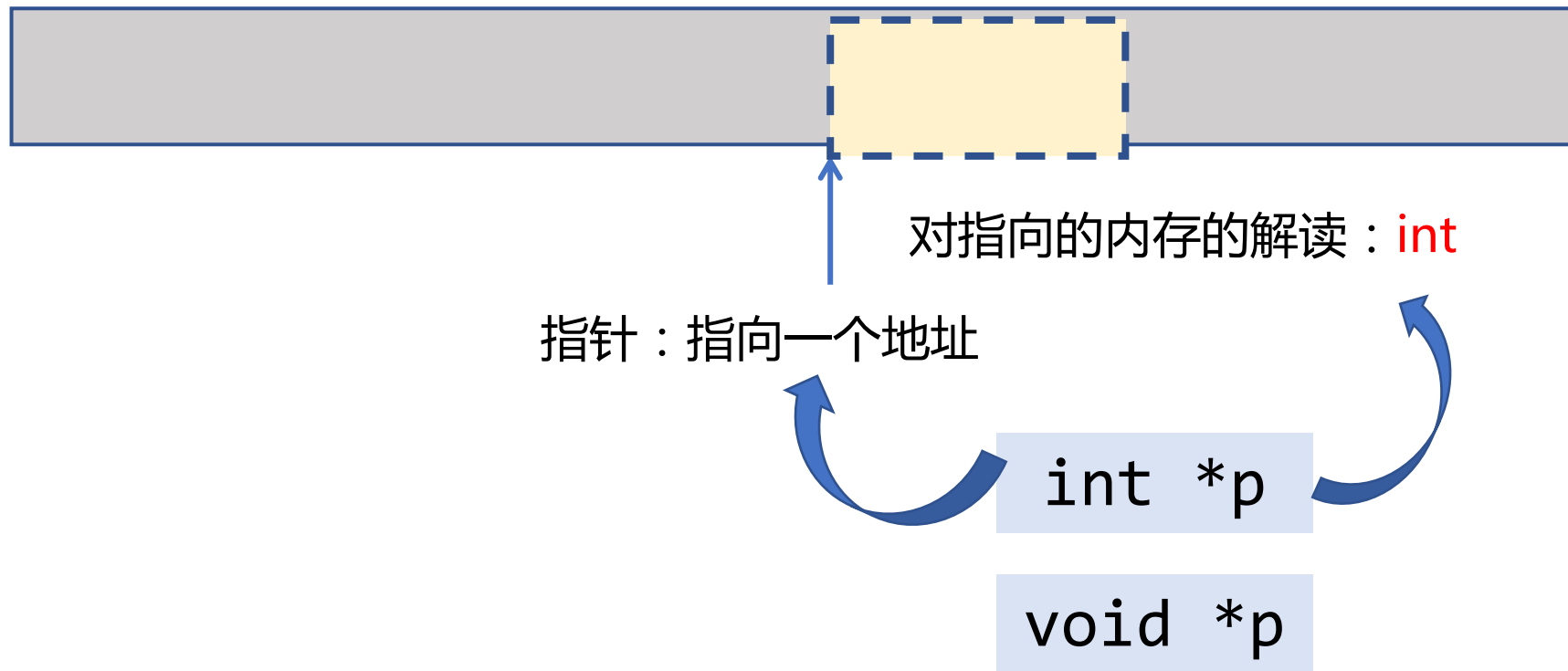


$x=1 \rightarrow$

$*p = \&x$
$*p = 1$

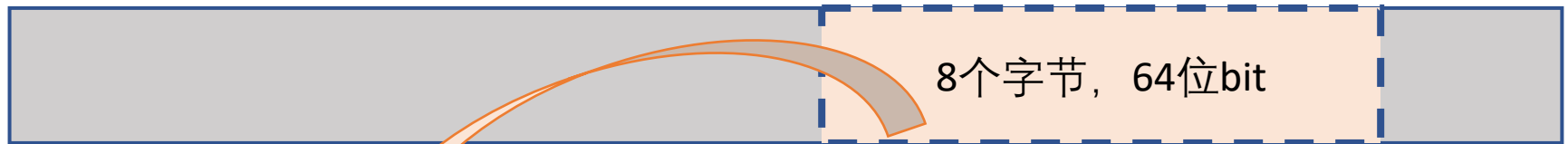
```
int main(int argc, char *argv[]) {  
    int *p = (void *) 1;    //OK  
    *p = 1;    //Segmentation fault  
}
```

# 内存



# 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\*4=64bit

# main, argc和argv

- 一切皆可取地址！

```
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]);
}
```

# 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）

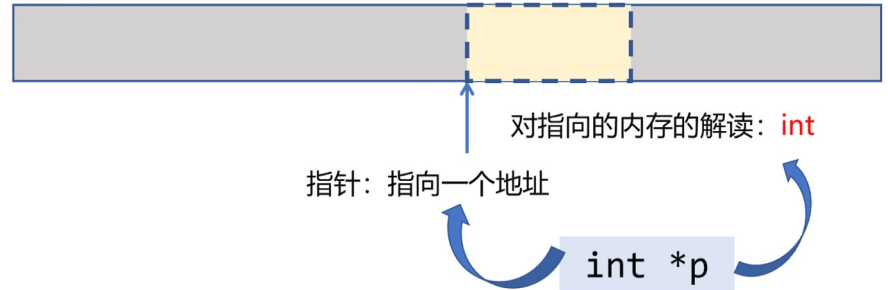
```
int main(int argc, char *argv[]) {  
    int (*f)(int, char *[]) = main;  
    if (argc != 0) {  
        char ***a = &argv, *first = argv[0], ch = argv[0][0];  
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);  
        assert(**a == ch);  
        f(argc - 1, argv + 1);  
    }  
}
```

```
$ ./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'
```



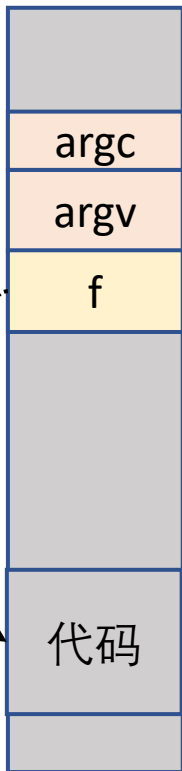
```
int main(int argc, char *argv[]) {  
    int (*f)(int, char *[]) = main;  
    if (argc != 0) {  
        char ***a = &argv, *first = argv[0], ch = argv[0][0];  
        printf("arg = \"%s\""; ch  
        assert(**a == ch);  
        f(argc - 1, argv + 1);  
    }  
}
```

## 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);  
    }  
}
```



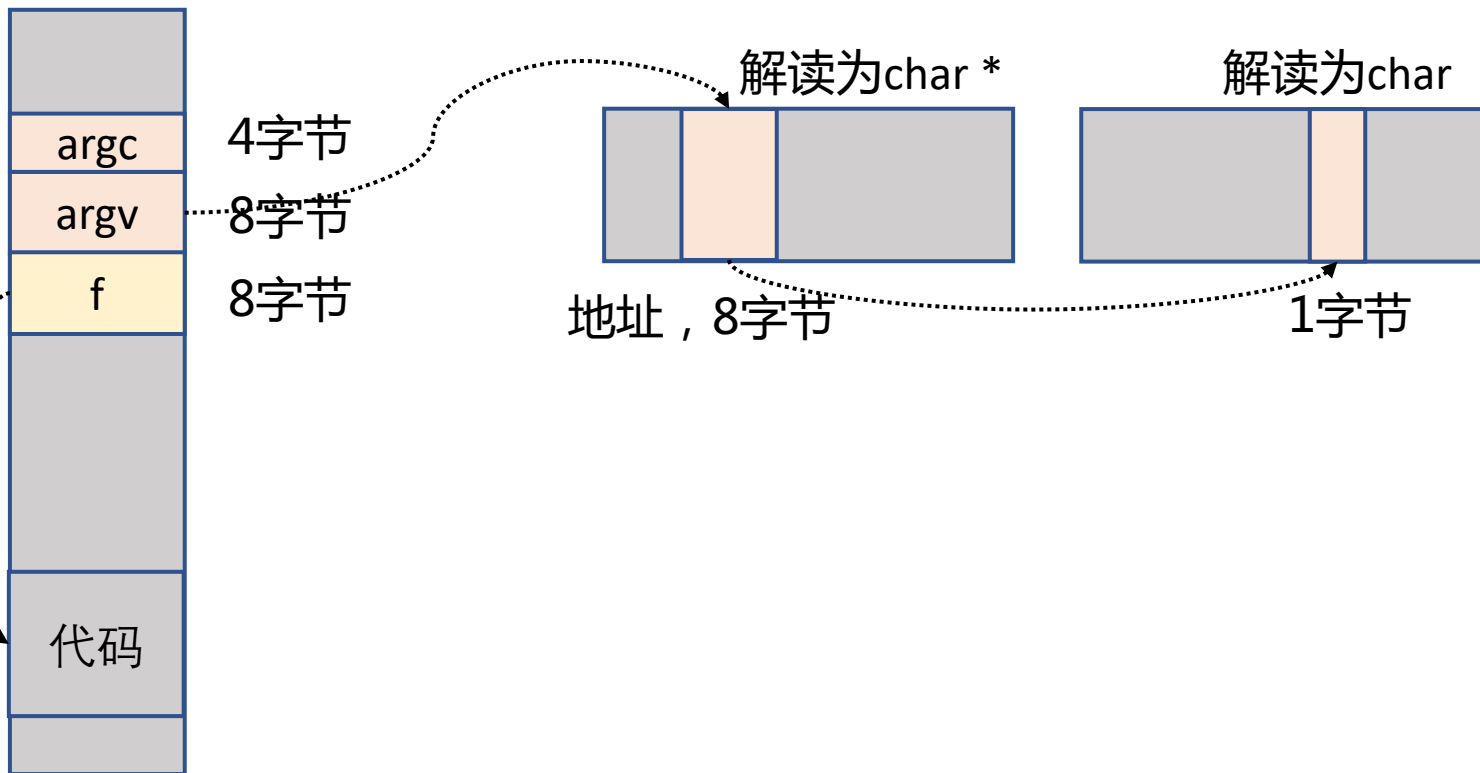
对指向的内存的解读: int  
指针: 指向一个地址  
int \*p

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

内存

堆栈

main



```

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);
    }
}

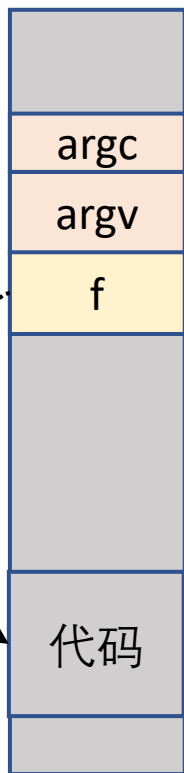
```



内存

堆栈

main



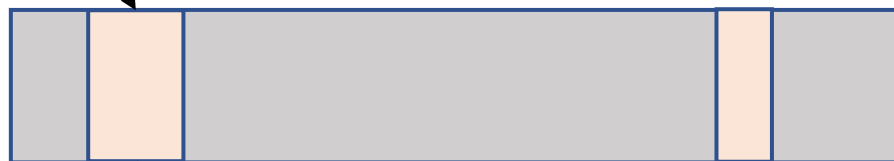
4字节

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);
    }
}

```

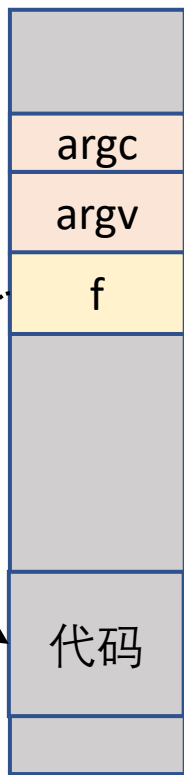


什么是argv+1 ?

内存

堆栈

main



4字节

8字节

8字节

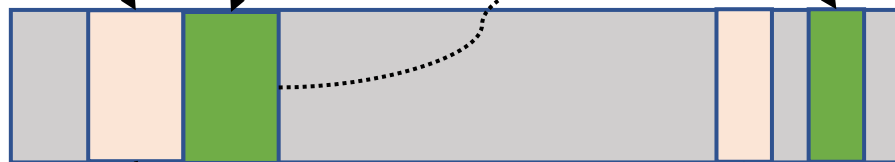
代码

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);
    }
}

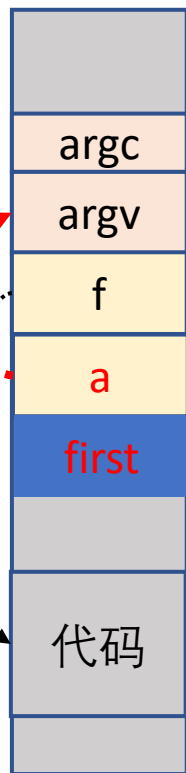
```



内存

堆栈

main



4字节

8字节

8字节

8字节

8字节

解读为char \*

argv+1、  
&argv[1]

地址，8字节

1字节

解读为char

按char解读

```

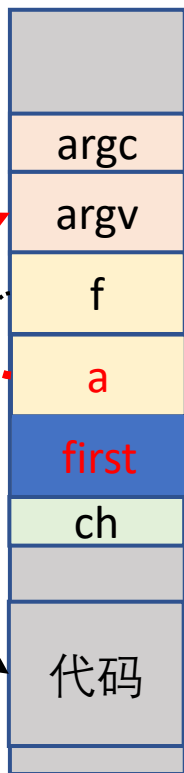
int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```

内存

堆栈

main



4字节

8字节

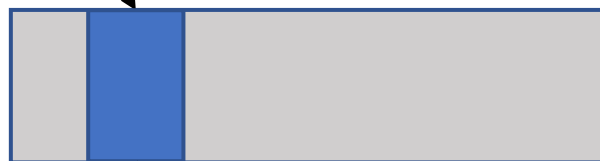
8字节

8字节

8字节

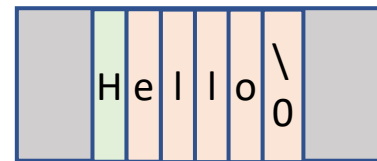
← H

解读为char \*



地址, 8字节

Hello



1字节

解读为char

按char解读

```

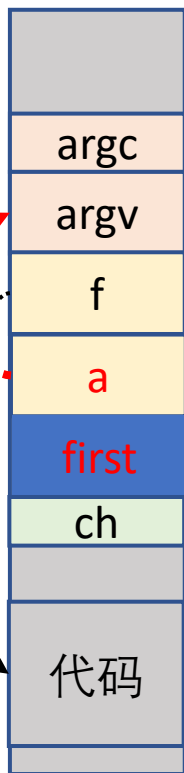
int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\\n", first, ch);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```

内存

堆栈

main 代码



4字节

8字节

8字节

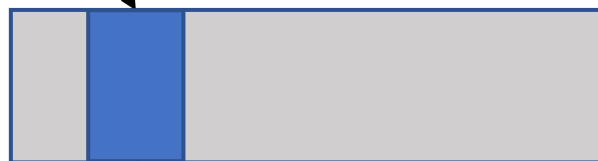
8字节

8字节

← H

解读为char \*

Hello



地址, 8字节

1字节

解读为char

按char解读

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

```

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", *a, *first);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```

↓

```

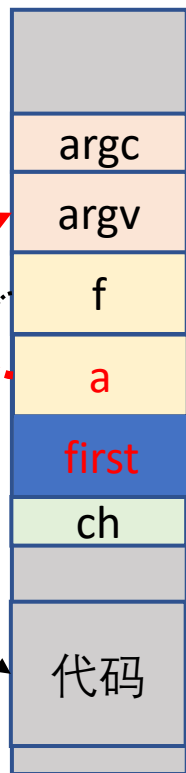
$ ./a.out 1 2 3 hello
arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'

```

内存

堆栈

main



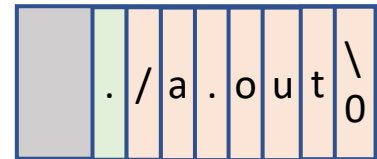
5

解读为char \*



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 = '.'

./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", *a, *first);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```

↓

```

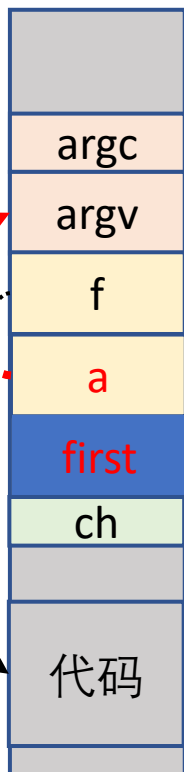
$ ./a.out 1 2 3 hello
arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'

```

内存

堆栈

main



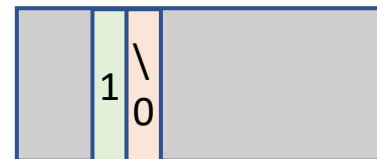
4

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);
    }
}

```

↓

```

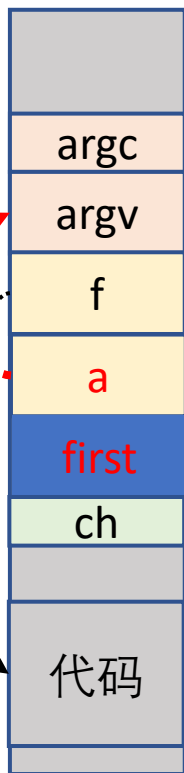
$ ./a.out 1 2 3 hello
arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'

```

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

```

```

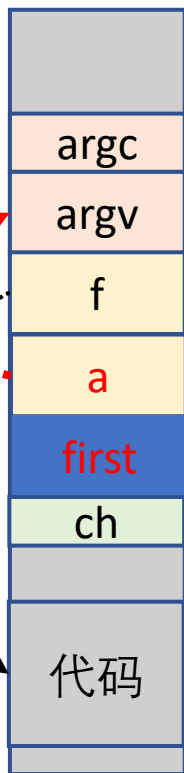
$ ./a.out 1 2 3 hello
arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'

```

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);
    }
}

```

↓

```

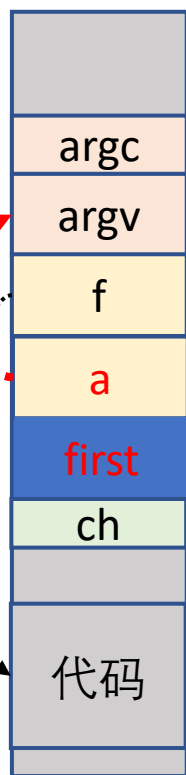
$ ./a.out 1 2 3 hello
arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'

```

内存

堆栈

main



1

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

./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

1字节

解读为char

	p	*p
<code>printptr(&amp;argc)</code>	a	argc值
<code>printptr(argv)</code>	c	d
<code>printptr(&amp;argv)</code>	b	c
<code>printptr(argv[0])</code>	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

---