# Program structure, preprocessing

王慧妍

why@nju.edu.cn

南京大学 软件学院 计算机软件研究所



```c
#include <stdio.h>

int main(void){



    return 0;
}
```

# 变量的生存期

```c
#include<stdio.h>
void f(){
    static int a;
    a ++;
    printf(" a = %d\n", a);
}
int b = 0;
int main(){
    int c = 5;
    printf(" b = %d, c = %d\n", b, c);
    f();
    f();
}
```

# What else is missing?

- include进来的是什么?
- #开头的编译预处理指令

```c
#include <stdio.h>

int main(void){



    return 0;
}
```

# 预处理

- #include
- #define

```c
#include<stdio.h>
#define N 100
int main(){
    int a;
    scanf("%d", &a);
    printf("%d\n", a * N);
}
```

看看预处理的结果?

```
gcc -E
```

# 三种典型预处理指令

- 文件包含
  - `#include`
- 宏定义
  - `#define`
- 条件编译
  - `#if, #ifdef, ifndef, #elif, #else, #endif`

✓ 指令以#开头

✓ 指令符号键可以插入空格或制表符

✓ 指令总是在第一个换行符结束（除非标记延续）

✓ 指令可以出现在程序任何地方（一般在开始）

✓ 指令可以与注释同行

# 宏定义

- 简单的宏（对象式宏）
  - #define 标识符 *替换列表*

  - 用于明示常量
    - #define TRUE 1
    - #define LEN 20
    - #define PI 3.14159

  - 其他用处
    - #define BOOL int
    - #define LOOP for(;;)
    - #define BEGIN {
    - #define END }

# 宏定义

- 带参数的宏（对象式宏）
  - #define 标识符(x1,x2,…) *替换列表*
  - 常作为类似简单函数使用
    - #define IS_EVEN(n) ((n)%2==0)
    - #define MAX(x,y) ((x)>(y)?(x):(y))
    - #define getchar() getc(stdin)
  - Macro能够类似定义了一种新的语言

# 宏定义

- 类似函数，确和函数调用并不完全一样
  - 考虑这种情况

```
#define MAX(x,y) ((x)>(y)?(x):(y))
……
n = MAX(i++, j)
```



```
n = ((i++)>(j)?(i++):(j))
```

- 更麻烦的是，宏定义导致的问题很难被发现！

# 宏-运算符

- 宏可以使用#和##两个运算符
  - #：将宏的参数转换为字符串，实现字符串化

```
#define PRINT(n) printf(#n " = %d\n", n)
int main(){
    int i = 10, j = 2;
    PRINT(i/j);
    return 0;
}
```

  - ##：粘合，实现字符串拼接

```
#define MK_ID(n) i##n
int MK_ID(1), MK_ID(2), MK_ID(3);
```

# 宏的括号

- 替换列表的括号，参数的括号
  - 好多括号= =

```
#define SCALE(x) (x*10)
```

```
#define SCALE(x) ((x)*10)
```

宏定义缺少圆括号会导致C语言中最让人讨厌的错误。程序往往仍然可以通过编译，而且宏似乎也可以工作，仅在少数情况下会出错。

```
#define min(x,y) (x)<(y)?(x):(y)
```

```
#define min(x,y) ((x)<(y)?(x):(y))
```

# assert

负责的宏，往往更麻烦，可以试试do{}while(0)或者大括号

```
#define assert(cond) if (!(cond)) panic(...);
```

- 注意特殊情况

```
if (...) assert(0); // 上面的assert对么?
else ...
```

```
#define assert(cond) \
do { \
   if (!(cond)) { \
      fprintf(stderr, "Fail @ %s:%d", __FILE__, __LINE__);
\
      exit(1); \
   } \
} while (0)

#define assert(cond) ({ ... }) // GCC
```

# Extra Clang Tools 17.0.0git documentation

## CLANG-TIDY - BUGPRONE-MULTIPLE-STATEMENT-MACRO



### bugprone-multiple-statement-macro

Detect multiple statement macros that are used in unbraced conditionals. Only the first statement of the macro will be inside the conditional and the other ones will be executed unconditionally.

Example:

```
#define INCREMENT_TWO(x, y) (x)++; (y)++
if (do_increment)
  INCREMENT_TWO(a, b);  // (b)++ will be executed unconditionally.
```


```
#define INCREMENT_TWO(x, y) (x)++; (y)++
if (do_increment)
  INCREMENT_TWO(a, b);  // (b)++ will be executed unconditionally.
```

# 宏的通用属性

- 宏的替换列表可以包括对其他宏的调用
  - `#define PI 3.14159`
  - `#define CRICLE(r) (PI*r*r)`
  - 宏的扩展遵循LIFO(last-in-first-out)

# 宏的通用属性

- 宏的替换列表可以包括对其他宏的调用
- 宏定义的作用范围通常到文件末尾
- 宏不可以被定义两遍，除非新旧定义一样的
  - 小的间隔上的差异允许，但是替换列表和参数需要一致
- 可以用#undef取消宏的定义
  - `#define N 10`
  - `#undef N`

# 预定义的宏

- ANSI C标准
  - `__FILE__`, `__LINE__`, `__DATA__`, `__TIME__`, `__STDC__`, etc.
- 部分扩展的宏
  - `__FUNCTION__`
  - `__COUNTER__`

# 带参宏定义 v.s. 函数调用

- 调用发生时间
- 参数类型检查
  - `#define MALLOC(n, type) ((type*) malloc((n) * sizeof(type)))`
- 参数空间分配
- 执行速度
- 代码长度

# 带参宏定义 v.s. 函数调用

更多可变长参数的宏定义请参见教材哦~

表 14.2 宏和函数的不同之处

| 属　性 | #define 宏 | 函　数 |
| --- | --- | --- |
| 代码长度 | 每次使用时，宏代码都被插入到程序中。除了非常小的宏，程序的长度将大幅增长 | 函数代码只出现于一个地方；每次使用这个函数时，都调用那个地方的同一份代码 |
| 执行速度 | 更快 | 存在函数调用/返回的额外开销 |
| 操作符优先级 | 宏参数的求值是在所有周围表达式的上下文环境里，除非它们加上括号，否则邻近操作符的优先级可能会产生不可预料的结果 | 函数参数只在函数调用时求值一次，它的结果值传递给函数。表达式的求值结果更容易预测 |
| 参数求值 | 参数每次用于宏定义时，它们都将重新求值。由于多次求值，具有副作用的参数可能会产生不可预料的结果 | 参数在函数被调用前只求值一次。在函数中多次使用参数并不会导致多个求值过程。参数的副作用并不会造成任何特殊的问题 |
| 参数类型 | 宏与类型无关。只要对参数的操作是合法的，它可以使用于任何参数类型 | 函数的参数是与类型有关的。如果参数的类型不同，就需要使用不同的函数，即使它们执行的任务是相同的 |

# 一个极端不可读的例子

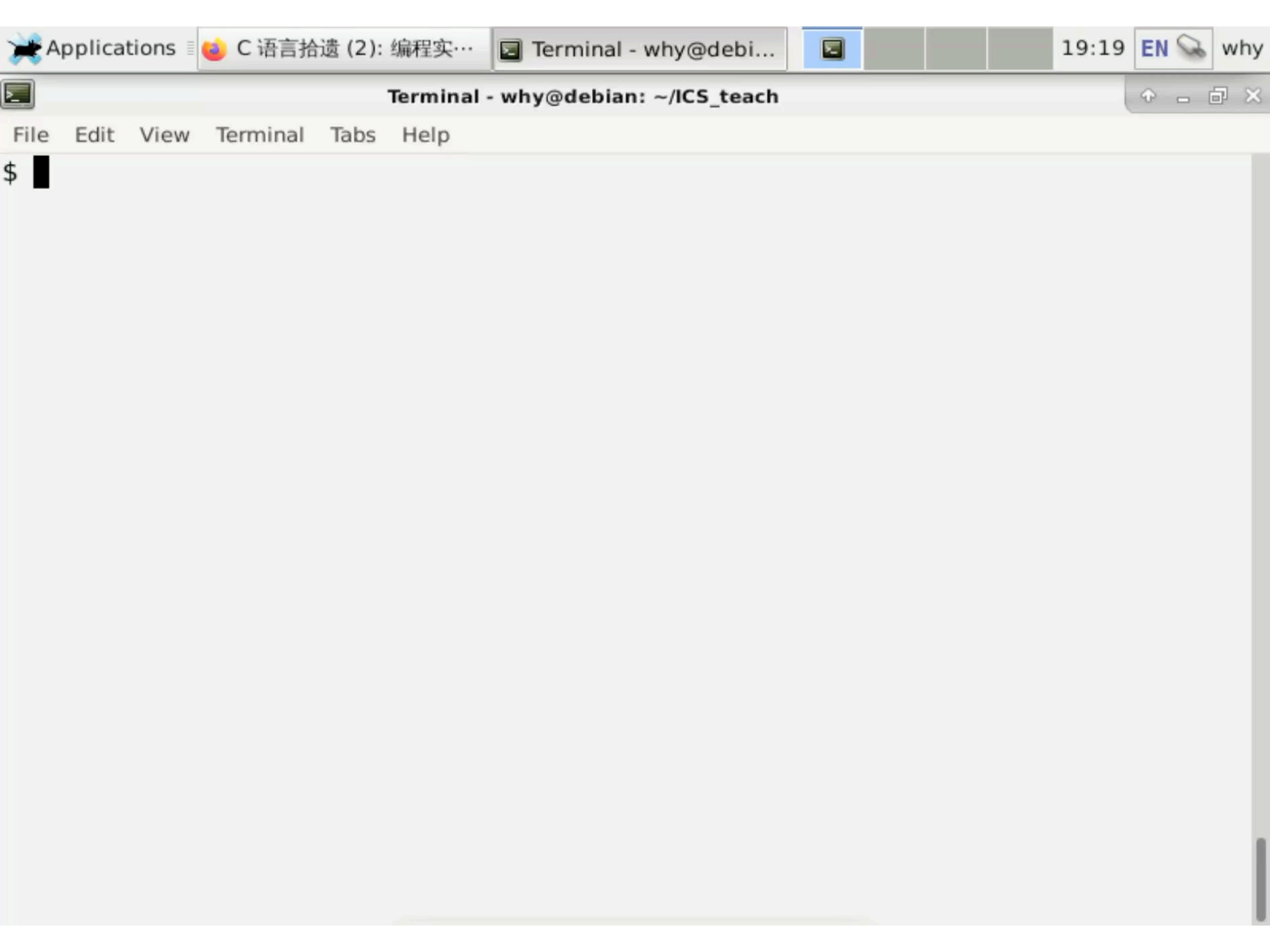
- IOCCC'11 best self documenting program
  - 不可读 = 不可维护

```
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
$

# 条件编译

```
#if    constant-expression
           statements
#elif  constant-expression
           statements
#else
           statements
#endif
```

# 常见practice

```
#if __STDC__
#ifdef(WIN32)
#ifdef(MAC_OS)
#ifdef(LINUX)
```

- 条件编译与宏的结合
  - 避免跨平台或跨模式的多版本重复工作
    - defined运算符：用于判断是否是定义过的宏

```
#if     defined(DEBUG)
            ……
#endif
```

```
#ifdef (DEBUG)
        ……
#endif
```

# 三种典型预处理指令

- 文件包含
  - #include
- 宏定义
  - #define
- 条件编译
  - #if, #ifdef, ifndef, #elif, #else, #endif

```c
#include <stdio.h>

int main(void){



    return 0;
}
```

# Program Structure

# 组织多文件程序

```
1   int max(int x, int y);
```

max.h

```
1   #include"max.h"
2
3   int max(int x, int y){
4       return x>y?x:y;
5   }
```

max.c

```
1   #include<stdio.h>
2   #include"max.h"
3
4   int main(){
5       int a = 5;
6       int b = 6;
7       printf("%d\n", max(a,b));
8   }
9
```

main.c

```
gcc -c main.c max.c -o main
```

# 编写大型程序

- 把程序划分成多个文件
  - 头文件
    - 一般包括宏定义，变量声明，函数原型
    - 惯例扩展名为.h
    - 全局变量：static, extern的区别
  - 源文件
    - 每个源文件包含程序的部分内容，主要是函数定义和变量定义
    - 某个源文件必须包含名为main的函数，作为程序的起始点

```c
int max(int x, int y);
```

```c
#include"max.h"

int max(int x, int y){
    return x>y?x:y;
}
```

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

int main(){
    int a = 5;
    int b = 6;
    printf("%d\n", max(a,b));
}

```

# 标准头文件结构

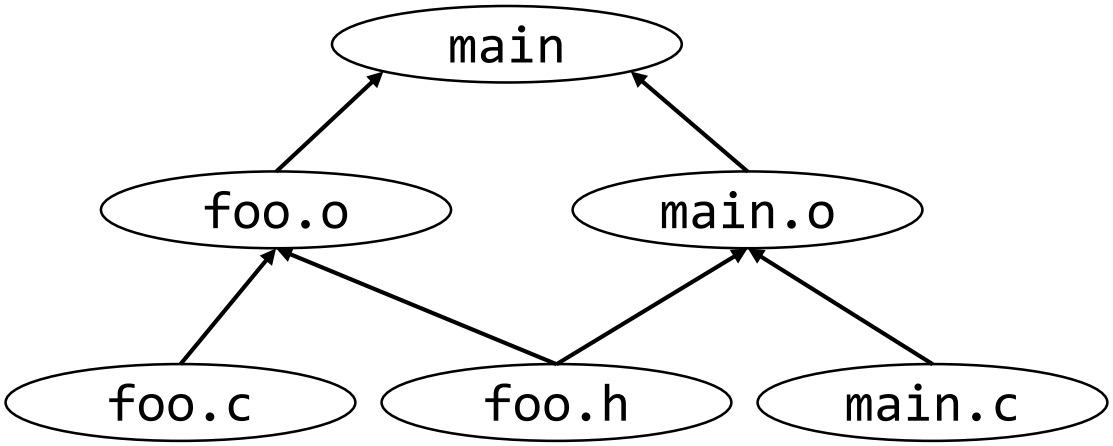
```
#ifndef XX
#define XX

#endif
```

#pragma once也能起到类似作用，
但是不是所有编译器都支持

# 构建多文件程序-Makefile

lec > make > M makefile

```
1  main: foo.o main.o
2      gcc -o main foo.o main.o
3  foo.o: foo.c foo.h
4      gcc -c -o foo.o foo.c
5  main.o: main.c foo.h
6      gcc -c -o main.o main.c
7  clean:
8      rm -f *.o
9  run: main
10     ./main
```



# CMake

```
cmake_minimum_required(VERSION 3.10)
project(cmake_demo)
# add_definitions("-Wall -Werror -fsanitize=address")
include_directories(../include)
aux_source_directory(../src SRC_LIST)
add_executable(cmake_demo ${SRC_LIST})
```

CMakeList.txt

.c, .h

cmake

Makefile

Make

./cmake_demo

lec > make > M makefile

```
1  main: foo.o main.o
2      gcc -o main foo.o main.o
3  foo.o: foo.c foo.h
4      gcc -c -o foo.o foo.c
5  main.o: main.c foo.h
6      gcc -c -o main.o main.c
7  clean:
8      rm -f *.o
9  run: main
10     ./main
```

# 更多思考

- 这也是为什么不在头文件里定义函数的原因
  - 两个 translation unit 同时引用，就导致 multiple definition

- 为什么C++可以实现函数重名（重载限制），而C不允许?
  - Name mangling
    - `_Z4funcid`

# I/O stream

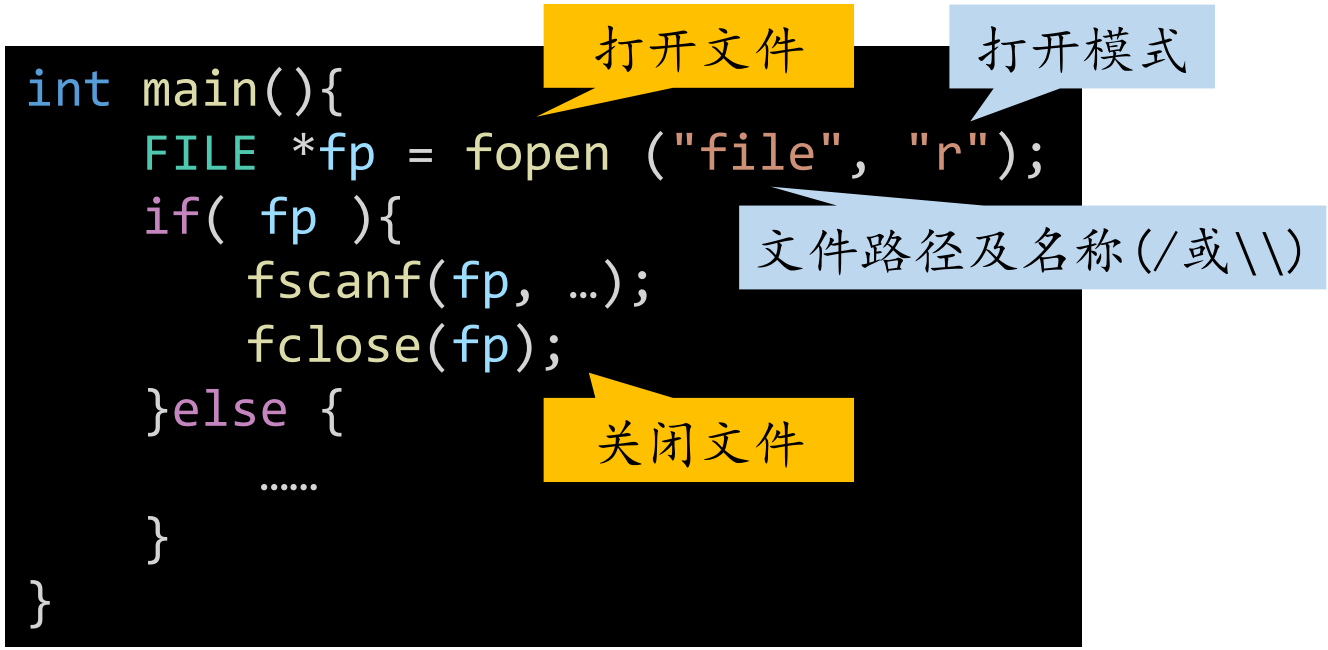
# 标准流和重定向

- <stdio.h>提供了3个标准流

| 文件指针 | 流 | 默认含义 |
| --- | --- | --- |
| stdin | 标准输入 | 键盘 |
| stdout | 标准输出 | 屏幕 |
| stderr | 标准误差 | 屏幕 |

# 文件操作

- 打开文本文件的标准代码



# 文本文件的模式字符串

| 字符串 | 含义 |
| --- | --- |
| “r” | 打开文件用于读 |
| “w” | 打开文件用于写（文件不需要存在） |
| “wx” | 创建文件用于写（文件不能已经存在） |
| “w+x” | 创建文件用于更新（文件不能已经存在） |
| “a” | 打开文件用于追加（文件不需要存在） |
| “r+” | 打开文件用于读和写，从文件头开始看 |
| “w+” | 打开文件用于读和写（如果文件存在就截去） |
| “a+” | 打开文件用于读和写（如果文件存在就追加） |

（p. 427）

# 文本文件输入输出函数家族

| 家族名 | 目的 | 可用于所有的流 | 只用于stdin和stdout |
|---|---|---|---|
| *getchar* | 字符输入 | fgetc, getc | getchar |
| *putchar* | 字符输出 | fputc, putc | putchar |
| *gets* | 文本行输入 | fgets | gets |
| *puts* | 文本行输出 | fputs | puts |
| *scanf* | 格式化输入 | fscanf | scanf |
| *printf* | 格式化输出 | fprintf | printf |

字符I/O

非格式化行I/O

格式化行I/O

# 二进制文本的模式字符串

| 字符串 | 含义 |
|---|---|
| “rb” | 打开文件用于读 |
| “wb” | 打开文件用于写（文件不需要存在） |
| “wbx” | 创建文件用于写（文件不能已经存在） |
| “w+bx”或“wb+x” | 创建文件用于更新（文件不能已经存在） |
| “ab” | 打开文件用于追加（文件不需要存在） |
| “r+b”或“rb+” | 打开文件用于读和写，从文件头开始看 |
| “w+b”或“wb+” | 打开文件用于读和写（如果文件存在就截去） |
| “a+b”或“ab+” | 打开文件用于读和写（如果文件存在就追加） |

（p. 427）

| 家族名 | 目的 | | |
|---|---|---|---|
| *fwrite* | 二进制输出 | | |
| *fread* | 二进制输入 | | |

# 输入输出函数（们）

- …scanf
  - int fscanf(FILE *__stream, const char *__format, ...)
  - int scanf(const char *__format, ...)
  - int sscanf(const char *__source, const char *__format, ...)
  - ……
- …printf
  - int fprintf (FILE *__stream, const char *__format, ...)
  - int printf (const char *__format, ...)
  - int sprintf (char *__stream, const char *__format, ...)
  - ……

# 更多：文件的刷新和随机读写

- `fflush(FILE *stream);`
- `fseek(FILE *stream, long offset, int from);`
- `void rewind(FILE *stream);`
- `int fgetpos(FILE *stream, fpos_t *position)`
- `int fsetpos(FILE *stream, fpos_t const *position);`

# End

- Bonus教学
  - 本节课内容不在教学计划内
  - 本学期授课结束（下节课为复习答疑课）