第1章-代码套路-华山剑法

代码风格形成风味 OR 套路后也被称呼为代码规范或者代码范式. 开发界流传至今范式很多, 其中以 Window 操作系统风格'武当流派', GNU Linux 操作系统极客风格'昆仑流派'最具代表性. 无论那种流派范式, 核心都在于有章可循自成方圆, 协助和提效工程开发. 本书是在昆仑流派基础上衍生, 也会尝试逐一分析其中范式缘由.

在如今这个互联网末法年代, 想一招鲜吃遍天, 可能充满荆棘和挑战, 更多需要更加系统训练培养自己专业力和自我兴趣或追求. 而在成长中也会时常遇到不熟悉领域, 一种有点意思方式是, 忘记以前了解的, 立即训练需要现在知道的, 先进去学后深究. 如果此刻你正好 C 第一本语法书, 数据结构书籍刚学完, 这里将会是很好拔高素材. 本章会围绕 针对 C 领域的范式, 带大家学习和训练 C 的起手套路. 不管怎么聊, 全文宗旨会一直延续, 写好代码首要条件是**写代码**! 然后从躬行思索**求简求美** ~

希望这本小册子带大家进入代码**写手**的世界. 尝试在自己动手中感受其中思路和设计!

1.1 缘起 main

很久很久以前进入一个被无数前辈巨擘称之为 main 函数. 回想其中经历的故事非常有趣. 这里只简单引述 C11 标准中规定的两种标准写法.

5.1.2.2.1 Program startup

1 The function called at program startup is named **main**. The implementation declares no prototype for this function. It shall be defined with a return type of **int** and with no parameters:

```
int main(void) { /* ... */ }
```

or with two parameters (referred to here as **argc** and **argv**, though any names may be used, as they are local to the function in which they are declared):

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

or equivalent;[10] or in some other implementation-defined manner.

2 If they are declared, the parameters to the **main** function shall obey the following constraints:

— The value of **argc** shall be nonnegative.

— **argv[argc]** shall be a null pointer.

— If the value of **argc** is greater than zero, the array members **argv[0]** through **argv[argc-1]** inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup. The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment. If the host environment is not capable of supplying strings with letters in both uppercase and lowercase, the implementation shall ensure that the strings are received in lowercase.

— If the value of **argc** is greater than zero, the string pointed to by **argv[0]** represents the *program name*; **argv[0][0]** shall be the null character if the program name is not available from the host environment. If the value of **argc** is greater than one, the strings pointed to by **argv[1]** through **argv[argc-1]** represent the *program parameters*.

— The parameters **argc** and **argv** and the strings pointed to by the **argv** array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.

main 是起点, 也是一切美好或梦魇的诞生点. C 学习历程可以从 Brian W. Kernighan / Dennis M. Ritchie 的 The C Programming Language 起步, 再到 C 标准文献, 随后熟悉操作系统. 此后遇到疑惑问题翻阅资料查看标准手册, 攀登在标准起源的山峰中, 大多数问题都会因上帝视角迎刃有解.

> 不知道这会有没有人想起 void main() {} or main() {} 写法呢?
> 对于 void main() {} 引述 C++ 之父 Bjarne Stroustrup 在其 FAQ 中说的

> The definition void main() { /* ... * / } is not and never has been C++,
> nor has it even been C.

> 编译器'老母亲'纵容了我们的"错误".
> 而 main() {} 最初出现在 K&R C 年代中, 那时候函数缺省定义(默认返回 int)不报警告.
> 当前 warning 甩过来的时候, 编译器老母亲醇厚的提醒应该重视和给予尊重.

一切才刚刚开始 -_- Good luckly

```c
#include <stdio.h>

int main(int argc, char * argv[]) {
    fprintf(stderr, "嗨! I'm here.\n");
    return 0;
}
```

1.2 括号布局

C 是极其强调细节的语言. 多数写手在用 C 开发的时候都会有点神经质. 难以接收铺张浪费. 恶心的是, 细节之处往往充满了妖魔神鬼, 让人'发狂'! 陆小凤通过一片落叶根蒂, 判断出自己绝不是西门吹雪对手. 同样一个 { } 布局就能见证你的起手式. 这门华山剑法 { } 遵循的如下规则和演示.

```
1' 左 { 右对齐
2' 右 } 上对齐
3' 右 } 是完整的一段代码才另起一行
```

```c
#include <stdbool.h>

// 1. for or while
for (;;) {      while (true) {
    ...
}               }

// 2. if ... else
if (true) {
    ...
} else {
    ...
}
```

对于 for (; ;) 和 while (true) 二者编译器最终优化生成的汇编代码是一样的, 但是推荐前者. 因为他更加简单优美.

对于 if else 为什么这么写, 先看下面对比.

```
if (true)
{
    ...
}
else if (false)
{
    ...
}
else
{
    ...
}
```

```
if (true) {
    ...
}
else if (false) {
    ...
}
else {
    ...
}
```

```
if (true) {
    ...
} else if (false) {
    ...
} else {
    ...
}
```

第一种写法是 Window 推崇的起手. 第三种是 Linux 上面常见写法. 第二种是两者的一种过渡. 对于上面三种范式, 都写过好久. 思索基于 C 精简特性, 这里推荐第三种. 更加紧凑飘逸! C 本身就是个精简的语言, 强调最小意外原则. 第一种直接 pass, 啰嗦的闸门一旦打开, 就难关上了. 第二种还是有点不极致. 同样对于 switch 语法推荐下面写法

```
switch (errno) {
case EINTR:
    ...
    break;
...
default:
    ...
}
```

switch 类比 C 中的标签都贴近作用域左边. switch brack 和 case 中语句对齐.

1.3 C 中三大战神 - 函数 § 帝释天

面向过程编程两个主旋律是内存(数据)结构和函数封装. C 函数开头一个 extern 关键词可能会搞晕不少同学. C 函数分为声明和定义两部分. 华山剑法中有 3 种套路写法. 函数声明: 确定函数地址; 函数定义: 确定函数地址和实现; 函数声明加定义: 确定函数地址和实现, 尝试限制作用域.

```
// 1' 函数声明
extern void hoge(void);

// 2' 函数定义 (返回值单独占一行, 用于强调, 标准套路)
void
hoge(void) {
    ...
}

// 3' 函数声明加定义 (extern 默认缺省定义)
[extern] void hoge(void) {
    ...
}
```

2' 中函数返回值另起一行为了突出函数(返回值)定义. 在同一行表示声明加定义. 隐含的一个细节是 hoge(void), 因为在 C 中函数声明 void hoge(); 表示的意思是函数 hoge 参数不确定(由 __cdecl 从右向左入栈方式决定). 加上 void 会在语法层面约束函数参数为空! 当我们写库头文件的时候, 下面两个函数声明, 表达出的意思就不一样. 华山剑法中为其添加的一种新假想语义.

```
// 函数声明 显示声明加上 extern
extern void hoge(void);

// 函数声明 缺省 extern 意图告知别人, 这是个 low level api 用的话推荐理解源码
void hoge(void);
```

编译层面二者没啥区别, 阅读层面传达的意图不一样, 前者是外部可以调用. 后者是外部不推荐调用, 调用的时候需要小心, 推荐了解源码细节. 当然了如果函数定义存在 inline 行为, 由于不同编译器对 inline 实现的差异, 为了照顾编译器差异, 声明时候可能必须要加上 extern, 用于确定函数定义地址. inline 和 register 很相似, 是书写者意愿表达, 具体看编译器大佬'心情'了! 请同学们多实战演练. 函数套路至关重要, 会陪伴咱们度过整个修真岁月.

同 extern 还有个关键字 static 特别神奇. 他描述变量和函数时, 表示在文件或函数内私有(静态)的. 请看下面演示代码 ->

```
static void heoo(int a[static 64]) {
    static int count;
    ...
}
```

局部代码中包含的 static 套路和潜规则剖析如下:

- 第 1 个 static 表示当前函数是私有的, 被包含的文件所有, 作用域是具体文件中这个函数定义行以下范围.

- 第 2 个 static 是 C99 出现的语法, 告诉编译器这个"数组"至少有 64 个数据单元, 您可以放心优化.
- 第 3 个 static 表示声明一个私有静态变量 count, 作用域属于 heoo 函数域.
- 另一个细节是 count 生命周期同未初始化全局的变量. 默认内存都是以 0 填充的. 即这里 count = 0 是缺省的.

> 科普 C 中
> **0, 00, 0x0, .0, '\0', '0', "0", NULL, false, EXIT_SUCCESS**
> 是什么鬼!

> 我们首先看某些实现版本源码中能找见的部分

```
#ifndef NULL
    #ifdef __cplusplus
        #define NULL 0
    #else
        #define NULL ((void *)0)
    #endif
#endif

#ifndef __cplusplus

#define bool    _Bool
#define false    0
#define true    1

#endif /* __cplusplus */

#define EXIT_SUCCESS 0
```

> 可以看出围绕 C 生态部分 NULL, false, EXIT_SUCCESS 都是宏. NULL 是个特殊的空指针 (void *)0, 后两个 false, EXIT_SUCCESS 具体定义是一样的数值零. 有时函数退出使用 exit(EXIT_SUCCESS); 或者 return EXIT_SUCCESS; 本书推荐采用前者 exit 写法(虽年少但沉稳老练). 开始的 0, 00, 0x0, '\0' 这几种都为零, 区别在于 00 是八进制, 0x0 是十六进制, '\0' 是字符常量. 零值都是一样的, 写者想表达出来的语义不同. .0 是 double 双精度的零, 对于 "0" 是个字符串常量, 等同于 char [2] { '0', '\0' } 这里也可以看出来 '0' 和 '\0' 不同字符常量 '0' == (int)48; 同样需要让人意外的是 C (sizeof '0' == 4)! 希望大家以后在 C 中看到这些数字, 能了然于胸, 乘云破雾.

> 顺带说说 **exit(EXIT_SUCCESS);** 写法居然木有警告?

```c
#include <stdlib.h>

// man 手册推崇套路范式
int main(void) {
    ...
    exit(EXIT_SUCCESS);
}
```

> 你是否好奇[**1' 为啥不 return EXIT_SUCCESS;**] 和 [**2' 而且编译器没有警告**]

> **1' 这要从小伙伴可能踩过坑说起.**
>
> 在 main 函数中 exit(EXIT_SUCCESS); 和 return EXIT_SUCCESS; 操作系统多数情况是相同的. 在 main 函数 return 会返回到系统的启动函数中, 会再次执行 exit 相关逻辑. 这就存在一个执行顺序依赖的问题了. 如果我们的主函数 return 了没有触发 exit, 那么二者就不一样. 看下面例子

```c
#include <stdio.h>
#include <stdlib.h>

int test(void) {
    puts("main not here");
    return EXIT_SUCCESS;
}

// $ gcc -g -Wall -O2 -nostartfiles --entry=test -o test.exe test.c
// $ ./test.exe
```

> 程序启动后, 退出时有些支持编译的系统会崩溃. 因为基础库准备的善后操作执行异常. 这时将 return 替换为 exit 就可以解决! 因为直接退出会走编译器为 exit 开通的善后流程. 为了安全起见有时候直接 exit 粗暴高效. C 很多依赖平台特性, 不保证每个平台都一致, 推荐熟悉一个主平台保持深挖和开放思维模式.

> **2' 编译器没有警告原因, 先看截取部分源码**

```c
#if defined(__GNUC__) || defined(__clang__)
    #define NORETURN(func) func __attribute__((noreturn))
#elif defined(_MSC_VER)
    #define NORETURN(func) __declspec(noreturn) func
#endif

// C11 中为其构造了新的关键字 _Noreturn
_Noreturn void suicide(void) {
    abort(); // Actually, abort is _Noreturn as well
}
```

> 脉络比较清晰了, 因为 exit 经过 _Noreturn 类似声明(标识), 永不返回, 因此编译器编译时就不再抛出无返回值警告. 分享 Over 哈哈哈. **赠送个打桩小技巧. 在协同开发时, 先快速设计好接口给对接方. 随后来个空实现!**

此刻再次为函数声明定个弱基调 😀

- 给其他模块用的函数, 推荐 extern 声明, 然后定义其实体
- 给自己模块用的函数, 推荐 static 静态(私有) 声明加定义
- inline 和各种缺省函数声明和定义, 理性使用

1.4 C 中三大战神 - 指针 - 达姿·奥利哈刚

C 中一个有争议的命题是一切皆内存. 而指针就是指向内存的魔法杖. C is free and unsafe. 至高奥义可能是

程序员是万能的

隐含的一层意思是你的程序你负责. 这其中最最强大功能就是指针, 他无所不能在当前的系统世界里. 本小节我们只演练华山剑法中 C 指针的写法范式. 首先看下面小段演示 Demo.

```
// 第1种: 传统写法, * 靠右
void *piyo, *hoge;

// 函数定义
void *
free(void *block) {
    ...
}

// 第2种: * 靠左, 模仿上层语言的类型
void* piyo, *hoge;

void*
free(void* block) {
    ...
}
```

微弱发现上面两种写法不太自然和统一. 由于 C 本身存在缺陷, 上面两种写法都不完美. 第 1 种, 函数定义的时候 * 号就不知道该放在那里了. 这种用法比较广, 但也不是最好的选择. 第 2 种写法, 多数是有过面向对象的编程经验, 想重温写哈 C. 这种写法在定义多个变量指针的时候基本就废了. 而且本身是变量指

针声明, 却被"幻想"成指针类型. 第 2 种写法强烈不推荐. 这里采用下面写法, 在追求自然和美过程中, 脚步不要停歇!

```
// 第3种: 变量声明, * 全部放中间
void * piyo, * hoge;

void *
free(void * block) {
    ...
}

// 补充说明, 多维指针, 函数指针声明
int ** arrs = NULL;
typedef void (* signal_f)(int sig);
```

关于指针范式基调主要如上, 简单补充关于 C 变量声明. C 目前共有 44 个关键字, 推荐命名要短小**精悍**. 强烈不推荐驼峰规则, 因为在远古时期 C 是不区分大小写所以很多库形成了小写传统. 而且 C 代码本身不容易读懂, 要懂的人会懂. 推荐遵从内核源码古法. 命名语义区分用 _ 分隔. C 也许不能让工资飞高, 但是可以让你在面对复杂问题的时候容易有细节思路 ~

小节最后不妨借花献佛. 引述<<C 语言问题>>书中让人豁然开朗, 关于 C 命名经典问题.

```
问: 如何判断哪些标识符可以使用, 那些被保留了?
答:
1' 标识符的 3 个属性: 作用域, 命名空间和链接类型.
[*] C 语言有 4 种作用域(标识符声明的有效区域): 函数, 文件, 块和原型. (第4种类型
    仅仅存在于函数原型声明的参数列表中)
[*] C 语言有 4 种命名空间: 行标(label, 即 goto 的目的地), 标签(tag, 结构, 联
    合和枚举名称), 结构联合成员, 以及标准所谓的其他"普通标识符"(函数, 变量, 类
    型定义名称和枚举常量). 另一个名称集(虽然标准并没有称其为"命名空间")包括了预
    处理宏. 这些宏在编译器开始考虑上述4种命名空间之前就会被扩展.
[*] 标准定义了 3 中"链接类型": 外部链接, 内部链接, 无链接. 对我们来说, 外部链接
    就是指全部变量, 非静态变量和函数(在所有的源文件中有效); 内部链接就是指限于
    文件作用域内的静态函数和变量; 而"无链接"则是指局部变量及类型定义(typedef)
    名称和枚举常量.

2' ANSI/ISO C 标准标识符标准建议规则:
规则 1: 所有下划线大头, 后跟一个大写字母或另一个下划线的标识符永远保留(所有的作
        用域, 所有的命名空间).
规则 2: 所有以下划线打头的标识符作为文件作用域的普通标识符(函数, 变量, 类型定义
        和枚举常量)保留(为编译器后续实现保留).
规则 3: 被包含的标准头文件中的宏名称的所有用法保留.
规则 4: 标准中所有具有外部链接属性的标识符(即函数名)永远保留用作外部链接标识符.
规则 5: 在标准头文件中定义的类型定义和标签名称, 如果对应的头文件被包含, 则在(同
        一个命名空间中的)文件作用域内保留.(事实上, 标准声称"所有作用于文件作用
        域的标识符", 但规则 4 没有包含标识符只剩下类型定义和标签名称了.)
```

1.5 C 中三大战神 - 宏 - 封神记·天

宏有点爽也有些疯狂. 需切记教条[能用 inline 内联, 就不要用宏!] 如果说指针是自由, 那宏就是噩梦. 我们从 **struct.h** 来认识这个偏执狂.

```c
#pragma once

#include <math.h>
#include <time.h>
#include <ctype.h>
#include <errno.h>
#include <float.h>
#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdint.h>
#include <limits.h>
#include <stdbool.h>
#include <inttypes.h>

//
// STR - 添加双引号的宏
// v        : 变量标识
//
#ifndef STR
#define STR_(v) #v
#define STR(v)  STR_(v)
#endif

#ifndef CONCAT
#define CONCAT_(X, Y) X ## Y
#define CONCAT(X, Y) CONCAT_(X, Y)
#endif//CONCAT

#ifndef LEN
//
// LEN - 计算获取数组长度
// a        : 数组变量
//
#define LEN(a)  ((int)(sizeof(a) / sizeof(*(a))))
#endif



#ifndef CMP_F
#define CMP_F

//
// cmp_f - left now node 比较 right input node 行为 > 0 or = 0  or < 0
// : int add_cmp(const void * now, const void * node)
//
typedef int (* cmp_f)();
```

```
#endif//CMP_F

#ifndef NEW_F
#define NEW_F

//
// new_f - 构建行为
// : void * rtree_new(void * node)
//
typedef void * (* new_f)();

#endif//NEW_F

#ifndef NODE_F
#define NODE_F

//
// node_f - 销毁行为
// : void list_die(void * node)
//
typedef void (* node_f)();

#endif//NODE_F

#ifndef EACH_F
#define EACH_F

//
// each_f - 遍历行为, node 是内部结点, arg 是外部参数; 返回值推荐 0 标识正确, -1 标识错误
// : int echo(void * node, void * arg) { return 0; }
//
typedef int (* each_f)(void * node, void * arg);

#endif//EACH_F

//
// DCODE - DEBUG 模式下的测试宏
// DCODE({
//     puts("debug test start ...");
// });
//
#ifndef DCODE
#  ifndef NDEBUG
#    define DCODE(code)  do code while(0)
#  else
#    define DCODE(code)
#  endif//NDEBUG
#endif//DCODE

//
// EXTERN_RUN - 函数包装宏, 声明并立即使用
```

```
// frun     : 需要执行的函数名称
// ...      : 可变参数，保留
//
#define EXTERN_RUN(frun, ...)                                           \
do {                                                                    \
    extern void frun();                                                 \
    frun (__VA_ARGS__);                                                 \
} while(0)

// PRINTF fprintf 包装操作宏. time_t x64 8字节 window %lld, linux %ld
#define PRINTF(stream, error, fmt, ...)                                 \
fprintf(stream, "[%"PRId64"]["#stream"][%s:%s:%d][%d:%s]"fmt"\n",       \
    time(NULL),                                                         \
    __FILE__, __func__, __LINE__, error, strerror(error), ##__VA_ARGS__)

#define POUT(fmt, ...)                                                  \
PRINTF(stdout, errno, fmt, ##__VA_ARGS__)

#define PERROR(error, fmt, ...)                                         \
PRINTF(stderr, error, fmt, ##__VA_ARGS__)

//
// PERR - 打印错误信息
// EXIT - 打印错误信息，并 exit
// IF   - 条件判断异常退出的辅助宏
//

#define PERR(fmt, ...)                                                  \
PRINTF(stderr, errno, fmt, ##__VA_ARGS__)

#define EXIT(fmt, ...)                                                  \
do {                                                                    \
    PERR(fmt, ##__VA_ARGS__);                                           \
    exit(EXIT_FAILURE);                                                 \
} while(0)

#define IF(cond)                                                        \
if ((cond)) EXIT(#cond)

//
// RETURN - 打印错误信息，并 return 返回指定结果
// val      : return 的东西. 填 NIL 标识 return void;
// fmt      : 双引号包裹的格式化字符串
// ...      : fmt 中对应的参数
// return   : val
//
#define RETURN(val, fmt, ...)                                           \
do {                                                                    \
    PERR(fmt, ##__VA_ARGS__);                                           \
    return val;                                                         \
} while(0)
```

```
#define NIL
#define RETNIL(fmt, ...)                                                \
RETURN(NIL , fmt, ##__VA_ARGS__)

#define RETNUL(fmt, ...)                                                \
RETURN(NULL, fmt, ##__VA_ARGS__)

// -1 是系统开发中最常见也算默认 error value 标识
#define RETERR(fmt, ...)                                                \
RETURN(-1  , fmt, ##__VA_ARGS__)

//
// 辽阔的夜，黝黑而深邃 ~
//
// 苏堤春晓
// 明·杨周
//
// 柳暗花明春正好，重湖雾散分林鸟。
// 何处黄鹂破暝烟，一声啼过苏堤晓。
//

// defined(__linux__) && defined(__GNUC__)
// defined(_WIN32)    && defined(_MSC_VER)
// defined(__APPLE__) || defined(__FreeBSD__) || defined(__OpenBSD__) || defined (__NetBSD__)
//

//
// 约定: BEST NEW VERSION 操作系统 Linux + 编译工具 GCC
//
#if defined(__linux__) && defined(__GNUC__)

# if defined(__x86_64__)
#  define ISX64
# endif

//
// With the new <stdbit.h>
// Endian macros (__STDC_ENDIAN_BIG__, __STDC_ENDIAN_LITTLE__, __STDC_ENDIAN_NATIVE__)

//
// 大小端检测 : ISBIG defined 表示大端
//
# if defined(__BIG_ENDIAN__) || defined(__BIG_ENDIAN_BITFIELD)
#  define ISBIG
# endif

# ifndef likely
#   define likely(x)   __builtin_expect(!!(x), 1)
# endif
# ifndef unlikely
```

```
#   define unlikely(x) __builtin_expect(!!(x), 0)
# endif

#endif

#ifndef UNUSED
#define UNUSED(parameter) if (parameter) {}
#endif//UNUSED

//
// With the new <stdbit.h>
// Endian macros (__STDC_ENDIAN_BIG__, __STDC_ENDIAN_LITTLE__, __STDC_ENDIAN_NATIVE__)
```

这里主要讲解华山剑法中宏的命名基本准则. 以下关于宏态度和演示, 多感受其中范式风味! 基调是**推荐不强求**, 求美求简 ❤ 就好. 上面 **struct.h** 封装一些函数指针类型主要为了后面写演示代码方便, 实战封装跟着具体的库走.

克制使用, 推荐用最清晰版本

```
#define STRUCT_H
#define RMRF_STR        "rm -rf \"%s\""
#define LOG_UINT        (2048u)
#define Q_INT           (1<<6)
#define ZERO_FLOAT      (0.000001f)

// 也能用, 不过宏名中的信息量没有上面全
#define N               (16)
```

上面是常量的宏命名好的示例, 统一大写. 而后 H, STR, UINT INT, FLOAT ... 让人很清晰知道宏的类型. 展示的第一个是头文件宏. 如果项目比较大可以在左边继续加上项目名称, 工程名称等等, 这就是命名空间由来. 其中字符串常量宏, 为了方便字符串拼接不用加 (). 数值常量宏防止意外拼接加了 (). 宏设计过程中, **当你不清楚会有什么意外会发生, 加括号是最保险的解决方案**.

函数宏另起一行写, 局部使用宏可以放一行写

```
// BZERO - 变量置零操作
#define BZERO(v)            \
memset(&(v), 0, sizeof(v))

//
// STR - 添加双引号的宏
// v    : 待添加双引号的量
//
#define STR(v) S_R(v)
#define S_R(v) #v
```

例如上面 BZERO(v); 可以独立存在, 加上 \ 另起一行. 对于 STR(v) 可以作为一个子语句, 直接写在一行就很清晰明了.

有些宏想被认为是函数, 那就小写

```
typedef volatile long atom_t;

#ifdef __GNUC__

#define atom_lock(o)        while(__sync_lock_test_and_set(&(o), 1))

#define atom_unlock(o)      __sync_lock_release(&(o))

#endif
```

写的人意图是希望 atom_lock 和 atom_unlock 被人当'函数'去使用, 使用了小写. 对于内置宏, **GNUC** 是标识 GCC 编译器, 表示当前用 GCC 编译项目会进入这个分支. 同样 _MSC_VER 是标识 M$ 的 CL 编译器. 简单的 project defined template 可以参照下面的 code. 做为开发人员, 推荐用最新的编译器. 因为保持活力和新鲜感很有趣, 一次新的尝试说不定就是一次机遇(多数就是坑 ...)

```
#if defined(__linux__) && defined(__GNUC__)

#elif defined(_WIN32) && defined(_MSC_VER)

#endif
```

> 命名规范简单总结

> a) 变量命名, 函数命名
> -> **推荐**全小写, **推荐**语义分隔使用 _
> -> 特殊是用大写字母缩写, 例如算法代码, 静态变量
> -> 宏中局部变量使用 $ 开头, 防止变量污染

```c
static struct files F;

static unsigned SDBMHash(const char * k) {
    register unsigned o,h = 0u;
    while ((o = *k++))
        h = o + h * 65599u;
    return h;
}

static atomic_int id = ATOMIC_VAR_INIT(1);

//
// EXTERN_TEST  - 单元测试宏，并打印执行时间
//                常量 CLOCKS_PER_SEC，它用来表示一秒钟会有多少个时钟计时单元
//                clock_t __cdecl clock(void) 该程序从启动到函数调用占用 CPU 的时间
// ftest        : 测试函数
// ...          : 可变参数
//
#define EXTERN_TEST(ftest, ...)                                                 \
do {                                                                            \
  int $id = atomic_fetch_add(&id, 1);                                           \
  printf("\n[test %d]. "STR(ftest)" run start ...\n\n", $id);                   \
  clock_t $start = clock();                                                     \
  extern void ftest (); ftest (__VA_ARGS__);                                    \
  clock_t $end = clock() - $start;                                              \
  printf("\n[test %d]. "STR(ftest)" run end difference clock %ld, time %lfms\n\n",  \
         $id, $end, (double)$end / (CLOCKS_PER_SEC / 1000));                    \
} while(0)
```

> b) 宏命名
> -> **推荐**全大写, 语义分隔使用 _
> -> 宏常量推荐 [名称]_[类型简写]
> -> 希望被当成函数使用的宏, 可以用小写命名

> c) 枚举声明, 类型声明
> -> C 枚举和 INT 宏常量很同质化, 默认类型 int. 有时候很显得多余, 能不用就不用 **用宏替代枚举**
> -> 类型声明 [name]_[类型缩写] 例如 start_f tstr_t 等, 类型区分将变得简单.

```c
#ifndef CSTR_INT

struct cstr {
    char * str;     // 字符串
    size_t cap;     // 容量
    size_t len;     // 长度
};

// CSTR_INT 构建字符串初始化大小
#define CSTR_INT    (1 << 7)

typedef struct cstr * cstr_t;

//
// cstr_declare - 栈上创建 cstr_t 结构
// cstr_free - 释放栈上 cstr_t 结构
// var      : 变量名
//
#define cstr_declare(var)               \
struct cstr var[1] = { {                \
    .str = malloc(CSTR_INT),            \
    .cap = CSTR_INT,                    \
} }

inline void cstr_init(cstr_t cs) {
    cs->len = 0;
    // 构建字符串初始化大小
    cs->cap = CSTR_INT;
    cs->str = malloc(CSTR_INT);
}

inline cstr_t cstr_new() {
    cstr_t cs = malloc(sizeof(struct cstr));
    cstr_init(cs);
    return cs;
}

inline void cstr_free(cstr_t cs) {
    free(cs->str);
}

#endif//CSTR_INT
```

到这基本把 C 的华山剑法的总纲讲的有小些了. 按照流派范式去写, 你会有更多的时间去学高深心法, 年岁久了再出来构建你自己的独孤草上飞. 同是不年轻的穷"屌丝", 要学会适应, 不流血那就流水. 都不简单, 也不难 ~

1.6 绝世好剑

万般皆自然, 一通都顺, 魔鬼在勤勉踏实的大道上ヾ(๑╹◡╹)ﾉ" 需要时间培养兴趣. 这里的绝世好剑指的是你的编程环境. 硬件方面要是可以的话买最快, 最美, 最便宜的套装. 软件方面, 我们只简单介绍昆仑流派.

昆仑流派

a) Install Best New Ubuntu Desktop LTS ISO
b) Install Best New GCC, GDB, VIM
c) Install Best New Visual Studio Code

Linux 是业界标杆, 简单高效优美. vi gcc make 能够搞定一切. 无数的一手资料, 强大的 man 手册. 题外话, 自己用老式 Linux 机器很快, 所以推荐一定要玩玩 Linux 平台, 其实什么环境都行(Window 系统技术进阶最曲折缓慢但操作上特别综合简单好用, Mac 非常舒服, Linux 很纯粹), 怎么舒服怎么来, 但都要醇厚精通 ~

剑指何方

个人感悟, 跨平台对于浮游生物而言是在浪费生命, 有些人单纯的用 Linux, 感觉非常纯粹, 越纯粹越强. Linux 是个回报率很高的平台. 但自从走上了 C 系语言之路. 真是天高地厚不知路长 ~ 岁月过的非常快 ... 书本内容很少更广阔世界留给各自兴趣去探索感悟吧 ~

1.7 夜太黑练剑好时光

聊的有些多, 细节部分需要自己亲身建构. 回想起 2013 年看<<自制编程语言>>那本书, 惊为天人. 感觉作者实力好强. 因为看不明白, 强行撸, 狂看猛打最后懂了点, 收益良多(虽然已经忘记了, 但却是另一番体验). 在编程的世界里, 不需要太多前缀, 初期只要 wo are 正在用手挥舞! 中后期需要兴趣和喜欢, 短期是出不了长期的成就 ~

如果一样只是为了, 更有意思的活着. 那么学起来就更随意了, 君子当善假于物! 熟悉工具, 实现想法, 自我积蓄德(包括武德)才 ~ 然后落叶生根, 想想也挺好.

如果你没有对象, 那就使劲敲代码

如果你觉得无聊, 那就跑步加看书

如果你有了家庭, 那就反思加打桩

如果你真不甘心, 那就呵呵萌萌哒

书归正转, 前面 **struct.h** 头文件中其实少了正规项目一个处理内存分配模块, 简单点说是我们对 malloc / free 等内存操作接口怎么处理的包装层. 常见的是下面这样模式

```
void * ptr = malloc(sizeof (struct type));
if (ptr == NULL) {
    // 业务兜底逻辑
}
free(ptr);
```

通过和 NULL 比较, 但实战中往往更加复杂, 因为内存不够时候, 程序在复杂系统中运行的情况往往是未定义的. 这个很考研工程师业务和技术理解, 也看系统整体规划. 这里简陋写一种简单粗暴方式方便大家感受下这类情况处理方法之一

```c
#pragma once

//
// 这是个非常简单粗暴 allocation 内存分配模块.
// 多数这类模块会和项目业务绑定，例如添加栈日志打印. 例如分配数据统计
//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// ation_check 内存检测并处理
inline void * ation_check(void * ptr, size_t size) {
    if (ptr == NULL) {
        fprintf(stderr, "check memory collapse %zu\n", size);
        fflush (stderr);
        abort();
    }
    return ptr;
}

//
// ation_malloc  - malloc 包装函数
// size     : 分配的内存字节
// return   : 返回可使用的内存地址
//
inline void * ation_malloc(size_t size) {
    return ation_check(malloc(size), size);
}

//
// ation_strdup  - strdup 包装函数
// str      : '\0' 结尾 C 字符串
// return   : 拷贝后新的 C 字符串
//
inline char * ation_strdup(const char * str) {
    if (str != NULL) {
        size_t n = strlen(str) + 1;
        return memcpy(ation_malloc(n), str, n);
    }
    return NULL;
}

inline char * ation_strndup(const char * str, size_t size) {
    if (str != NULL) {
        // @see https://stackoverflow.com/questions/66346502/which-is-most-standard-strnlen-or-s
        // POSIX 标准真是良心. 好的标准往往容易有好的生态. 而不需要程序员和土匪似得东抢西偷.
        size_t n = strnlen(str, size);
        char * dup = ation_malloc(n+1);
        dup[n] = 0;
```

```
        return memcpy(dup, str, n);
    }
    return NULL;
}

//
// ation_calloc  - calloc 包装函数
// num      : 数量
// size     : 大小
// return   : 返回可用内存地址, 并置 0
//
inline void * ation_calloc(size_t num, size_t size) {
    return ation_check(calloc(num, size), size);
}

//
// ation_realloc - realoc 包装函数
// ptr      : 首地址, NULL 等同于 malloc
// size     : 重新分配的内存大小
// return   : 返回重新分配的新地址
//
inline void * ation_realloc(void * ptr, size_t size) {
    return ation_check(realloc(ptr, size), size);
}
```

对于上面 alloc.h 中 ation_check 函数思路, 在内存不足时, 直接 abort. 不知道有没有朋友会问为什么不用 exit? 思考的出发点是这样的, 当出现申请内存不足的时候. 笼统的概括有两种场景:

- 1' 申请大内存,
- 2' 申请小内存.

如果是 1' 申请大内存, exit 退出是可行, 并且还会执行 atexit 相关函数.

但如果是 2' 申请小内存, 小内存都耗尽, 那么后续都将是未定义. 综合 1' 和 2' 加上这是个学习项目定位最终
使用了 abort. 当内存不足的时候, 直接'休息', 粗暴有效的上车.

对于上面真的内存不足去打印信息 fprintf 也是未定义, 大多数情况没问题吧. 以上简单直接使用操作系统能力, 也可以采用了近代软件编程中免费午餐 jemalloc 来包装自己的 alloc.h 层. jemalloc 科普可以搜查资料, 对于如何编译成静态库并使用, 可在 jemalloc github 主页获取官方方法. 整体 alloc.h 是可插拔的, 这也是程序封装一个好技巧, 好用方便.

:0 初期练习的人体内要感受到兴奋 和 颤抖 ~ 原来变强就是这么简单, 系统临摹和训练. 啊哈. 此刻只想说三个字, 这很 cool ~ 享受 code.

> 心田的草 ——<<@

也许 - 汪国真

也许，永远没有那一天
前程如朝霞般绚烂
也许，永远没有那一天
成功如灯火般辉煌
也许，只能是这样
攀援却达不到峰顶
也许，只能是这样
奔流却掀不起波浪
也许，我们能给予你的
只有一颗
饱经沧桑的心
和满脸风霜

C 是个传统古老编程领域, 当你想入门时候, 可能需要读更多好书, 记录更多笔记, 写更多代码, 让他成为你的额外母语, 感受编程细节的魅力.

——<— *练剑的你, 终有一天将坑满天下*


笑傲江湖
金庸