HZ BOOKS
华章科技

C/C++
技术丛书

C高级编程

基于模块化设计思想的C语言开发

吉星 著

Professional C With Modular Design Method

- C语言模块化设计和编程的典范之作，高度呈现模块化设计的思想与精髓，系统总结模块化的系统设计方法
- 以大量可复用的C工程代码为依托，深入地讲解了C语言的核心技术和重要模块，以及如何用模块化的方法进行大规模工程实践

机械工业出版社
China Machine Press

C/C++
技术丛书

C高级编程

基于模块化设计思想的C语言开发

Professional C With Modular Design Method

吉星 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

C 高级编程：基于模块化设计思想的 C 语言开发 / 吉星著．—北京：机械工业出版社，2016.5
（C/C++ 技术丛书）

ISBN 978-7-111-53641-3

I. C… II. 吉… III. C 语言－程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2016）第 086963 号

C 高级编程：基于模块化设计思想的 C 语言开发

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：李 艺	责任校对：殷 虹
印 刷：三河市宏图印务有限公司	版 次：2016 年 5 月第 1 版第 1 次印刷
开 本：186mm×240mm 1/16	印 张：33.25
书 号：ISBN 978-7-111-53641-3	定 价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066　　投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259　　读者信箱：hzit@hzbook.com

版权所有 • 侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

Preface 前　言

为什么要写这本书

因为工作原因，在算法优化、底层驱动、嵌入式系统设计等方面的软件编程时，一直使用 C 语言，而且很难有其他“更好”的选择。一方面，工作内容在客观上决定了无法利用更高级语言；另一方面，相对其他语言，在上述工作领域中持续使用 C 语言，使得工作效率更高（结合必要的 shell 脚本）。因此对于那些初入上述工作领域的工程师，我始终推荐 C 语言。通过本书，希望将个人的开发总结作为示例，给予新人作为参考。

C 语言是一种比较早期的高级语言，其本身是模块化的，这使得通过 C 语言比较容易实现面向电子、计算、自控系统自身的模块化设计。目前更多的软件设计并非针对电子、计算、自控系统本身，例如，一个企业管理软件、一个网站商城界面等。这些软件设计，是基于应用者的思维，或者说人类正常思维模式而展开的。由此，这类设计使用面向对象语言会非常方便，但却导致过多关注计算机编程的教育，忽视了面向模块化编程方法的讲解。因此，本书将模块化系统设计的个人总结与 C 语言的讨论融合。希望本书能抛砖引玉，让上述工作领域的读者更好地关注与思考面向系统本身的设计方法。

本书特色

在本书写作的过程中，使用了个人工程代码库中的原型，并尽可能保证这些代码有一定的应用价值。为了在有限的章节尽可能给出一个较为完整的代码集合，因此，章节之间的代码存在一定依赖性，即，前序代码形成的模块，会被后续章节中所讨论的代码利用。

为了让工程经验欠缺的新人对 C 语言开发有更好的感性认识，本书在讨论问题和介绍代码中穿插了很多个人观点，这些观点并不是理论，也不一定是行业共识，只是从一个侧面的经验之谈，希望对读者有参考价值。

读者对象

- ❑ 电子、自控、计算机等相关专业的高年级本科、研究生
- ❑ 算法设计与优化工程师
- ❑ 嵌入式系统开发工程师
- ❑ 底层、中间件子系统开发工程师
- ❑ 其他对 C 语言编程、模块化系统设计感兴趣的人员

如何阅读本书

本书共九章，从 C 语言自身，一直探讨到（进程）模块之间的共享与通信。前八个章节，更多是工程和具体代码设计的讨论，而最后一个章节则是系统分析与系统设计方法的讨论。对于期望、正在从事系统整体规划、构架、设计的读者，建议首先了解最后一章内容，而对于欠缺系统分析经验的新进工程师，则建议从第 1 章开始阅读，同时建议对书稿中的代码进行上机验证，在执行反馈中了解本书的观点，并进行修正，形成自身工程代码库。

勘误和支持

由于水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正，期待能够得到你们的真挚反馈，在技术之路上互勉共进，我的邮箱是 zsu_lucky@163.com。

致谢

感谢教育、指导、帮助、支持过我的老师、朋友及家人，使得我能持续多年在所喜爱的技术领域进行工作。

感谢机械工业出版社华章公司的杨福川和高婧雅，始终支持我的写作，是你们的鼓励和帮助引导我顺利完成这本书稿。

最后，特别感谢杨尚丽对本书的文句审核以及赵瑞源对本书代码的验证。

吉星

2016 年 3 月

试读结束：需要全本请在线购买： www.ertongbook.com

Contents 目　录

前　言

第1章　C语言的探讨 ……… 1

1.1　C的编译链接与文件引用 ……… 3

1.1.1　一个小程序 ……… 3

1.1.2　链接与文件引用 ……… 5

1.2　函数、数据与作用域 ……… 8

1.2.1　全局函数与局部函数 ……… 8

1.2.2　数据与数据的类别 ……… 9

1.2.3　数据存储空间 ……… 11

1.3　类型与操作 ……… 17

1.3.1　基础类型及其操作和重定义 ……… 17

1.3.2　结构体类型 ……… 20

1.3.3　指针常量、指针与连续空间 ……… 21

1.3.4　函数接口类型、可变参类型和执行跳转 ……… 26

1.4　预处理操作 ……… 31

1.4.1　C语言的词法与预处理 ……… 31

1.4.2　宏判断的应用 ……… 37

1.4.3　宏定义与模板函数 ……… 40

1.4.4　预处理的杂项 ……… 43

1.4.5　宏与代码的自动化构建 ……… 47

1.5　小模块与函数内的模块化 ……… 52

1.5.1　参数判断小模块 ……… 52

1.5.2　goto与函数内的模块化 ……… 54

1.6　结束语 ……… 63

第2章　标准库、自有基础库与delog模块 ……… 64

2.1　标准库 ……… 65

2.1.1　assert.h、errno.h ……… 66

2.1.2　setjmp.h跨函数的跳转 ……… 67

2.1.3　stdarg.h ……… 69

2.1.4　stdio.h ……… 74

2.1.5　stdlib.h ……… 82

2.1.6　string.h ……… 86

2.1.7　time.h ……… 90

2.2　构建自有基础库 ……… 91

2.2.1　基础操作 ……… 93

2.2.2　char的表 ……… 99

2.2.3　UTF-8的基础表 ……… 106

2.2.4　慢一点的字符串操作 ……… 112

2.3　delog模块 ……… 118

2.3.1　实体模块 ……… 118

2.3.2 delog 模块利用的系统库函数 …… 119
2.4 结束语 …… 133

第 3 章 make、工具与文档组织 …… 134

3.1 依赖与 make …… 135
3.2 自有工具库 …… 146
3.2.1 lex/flex 的入门应用 …… 147
3.2.2 参数配置的子模块 …… 153
3.2.3 自己构造自己的小工具 …… 175
3.3 工程文档的组织 …… 190
3.3.1 makefile 的补充设计 …… 190
3.3.2 模块目录与工具 …… 193
3.3.3 整体的工程目录 …… 203
3.4 结束语 …… 211

第 4 章 抽象逻辑与虚拟模块、索引模块 …… 213

4.1 抽象与宏 …… 214
4.1.1 抽象的描述 …… 214
4.1.2 抽象的对象与操作 …… 217
4.1.3 抽象的函数 …… 221
4.2 虚拟模块 …… 228
4.2.1 抽象类型的定义 …… 228
4.2.2 抽象操作和模板函数的组织方式 …… 232
4.2.3 模板函数的引用 …… 234
4.3 索引模块 …… 241
4.3.1 索引链及其基础操作 …… 243
4.3.2 索引模块的构建 …… 246
4.3.3 索引节点的存储单元 …… 249
4.4 结束语 …… 252

第 5 章 空间资源的组织 …… 253

5.1 资源的申请与利用 …… 254
5.1.1 动态空间的获取 …… 254
5.1.2 基于 POSIX 的共享空间 …… 259
5.1.3 基于 POSIX 的信号量 …… 263
5.2 连续空间的组织 …… 268
5.2.1 连续空间的组织信息 …… 270
5.2.2 通用数据空间类型 _BUF 与 _P …… 272
5.3 两个空间管理模块 …… 278
5.3.1 jx_buf 模块 …… 278
5.3.2 jx_sharebuf 的子模块 …… 301
5.3.3 jx_sharebuf 共享空间管理模块 …… 309
5.4 结束语 …… 320

第 6 章 数据的集合化组织 …… 321

6.1 集合化空间的组织 …… 323
6.1.1 集合的元素节点 …… 324
6.1.2 集合的组织 …… 325
6.1.3 节点空间的组织 …… 329
6.2 虚拟模块 jx_sets …… 331
6.2.1 jx_SetsType.h 和 jx_SetsDef.h …… 332
6.2.2 模板函数头文件 jx_SetsTemp.h …… 334
6.2.3 模板化虚拟模块的实例 …… 354
6.3 集合化空间的扩展 …… 359
6.3.1 散列集合化空间的组织方式 …… 359

6.3.2 散列集合化空间的操作 ········· 363

6.3.3 散列集合化空间的测试 ········· 370

6.4 结束语 ········· 373

第 7 章 复杂的数据集合化 ········· 375

7.1 树集合化空间 ········· 376

7.1.1 树的结构与基础操作 ········· 377

7.1.2 树集合化空间管理模块 ········· 382

7.1.3 相对复杂的树操作 ········· 392

7.2 有向关系集合化空间 ········· 399

7.2.1 有向关系结构与基础操作 ······ 400

7.2.2 关系集合化空间的基础操作 ··· 404

7.3 有向图的集合化空间 ········· 416

7.3.1 模块的类型和定义 ········· 418

7.3.2 模板函数 ········· 424

7.3.3 可配置的遍历与搜索 ········· 432

7.4 结束语 ········· 441

第 8 章 进程与通信 ········· 442

8.1 多进程的创建 ········· 443

8.1.1 使用 fork 创建进程 ········· 444

8.1.2 创建新会话 ········· 447

8.1.3 调整文件的关联 ········· 449

8.2 socket 方式的进程间通信 ········· 452

8.2.1 socket 通信 ········· 452

8.2.2 jx_socket 模块 ········· 456

8.2.3 会话与测试 ········· 469

8.3 基于共享空间的进程间通信 ········· 472

8.3.1 基于共享空间的队列模式 ······ 473

8.3.2 基于共享空间的多端口模式 ········· 481

8.3.3 多端口模式的应用场景 ········· 492

8.4 结束语 ········· 496

第 9 章 模块化的系统设计 ········· 497

9.1 系统与模块 ········· 498

9.1.1 什么是系统 ········· 498

9.1.2 什么是模块 ········· 500

9.1.3 模块化与关联 ········· 503

9.2 模块化的分析与设计方法 ········· 506

9.2.1 角色与任务 ········· 506

9.2.2 框架与层次 ········· 509

9.2.3 协同对接与系统整合 ········· 511

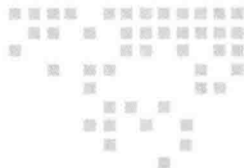
9.3 C 语言与模块化 ········· 512

9.3.1 进程与模块 ········· 513

9.3.2 模块封装与接口协议 ········· 516

9.3.3 各种模块与抽象、复用 ········· 519

9.4 结束语 ········· 521



第 1 章 Chapter 1

C 语言的探讨

本章主要针对那些刚刚离开校园，准备参与基于 C 语言设计的项目工程，从事 C 语言程序开发的初级工程师；或已初步学习了 C 语言的语法知识，可独立编写一些小的 C 语言程序，但对 C 语言的设计方法和特点并未全面掌握的初级程序员。

本章（其实包括本书）会有很多观点与传统教科书的描述内容存在差异。这种差异并不是对已有教科书部分内容的否定，更不是对辛勤的教育工作者们的否定。这种差异来源于教育与实际开发工程所在的环境差异以及程序设计任务目标的差异。

单纯依靠传统教科书的内容并不能有效支撑实际工程的开发，同时有些从工程开发角度所关注的内容被传统教科书忽略。因此本章从工程实践角度出发，探讨 C 语言的一些概念、设计内容和方法，以便初级程序员对 C 语言及其利用有进一步的了解。而更全面地掌握需要程序员在实际工程开发中逐步地感悟与积累。

本章受篇幅限制，仅仅针对 C 语言在编译链接、函数、数据类型、指针、预处理操作等几个方面展开讨论。同时这些讨论的内容，大多会出现在本书后续章节的程序设计中，因此也可将本章看作对本书后续章节讨论内容的铺垫。

本章不会（也不可能）将所有 C 语言以及关联的内容全部展开，而是希望借助一些举例和讨论，引出相关概念和知识点，便于初级程序员有针对性地查找相关资料以做更深入的了解，本书后续对此类情况，将简称为“参照相关资料”。相关资料中，至少包含以下 4 方面内容：

- C 语言国际标准；
- 你所使用编译器的产品手册；
- 你所使用编译器的基础库手册；
- 你所使用操作系统提供的 C 语言接口库手册。

注意 以上四方面，一切以 C 语言国际标准为基准。或许 3 ~ 5 年的一个 C 语言应用阶段，你一直使用同一个编译器针对同一个目标操作系统进行开发，但你不能保证在随后 10 年、20 年的开发工作中所用的编译器和针对的目标操作系统始终不变。对于有出入的内容，需要非常注意，在系统原型设计阶段，尽可能地回避编译器的特性。在系统优化阶段，再针对特定目标系统，借助编译器 / 操作系统接口库的特性内容来提升你的程序性能。对不属于 C 语言标准库的内容，则应尽可能地选择那些符合国际标准的协议、规则、规范（如 POSIX 协议）的部分。

在展开本章讨论之前，围绕 C 语言的开发，此处给出一些本书作者（本书后续简称“我”）的个人建议。

建议 1：

区别于那些“更高级”的计算机编程语言，C 语言的设计，从你刚开始入行时就应该有一个意识：“基于 C 语言的程序开发项目，应当分为系统原型开发阶段和目标平台优化阶段”，前者在利用 C 语言本身，后者在发挥特定目标平台的优势。前者关注系统的内在逻辑，后者关注平台的具体特性。但不同的目标平台可能差异较大，如果需要发挥它们的优势，可能存在特定的数据组织策略，这些策略需要在前期阶段进行逻辑验证，因此这两个阶段也不是完全可隔离的。其他“更高级”的语言，并不太关注硬件、平台特性，而 C 程序员应当多了解目标系统（硬件以及操作系统）的运行机理。

建议 2：

C 语言的设计开发，尽可能地在类 UNIX（UNIX 的各种演化版本）或 Linux（参照 UNIX 在 PC 上实现的操作系统，其不属于类 UNIX）下而非 Windows 平台之上进行，并优先学习这类操作系统及面向这类操作系统开发的技巧。C 语言诞生在 UNIX 之上，最初的目的又是为了设计 UNIX 操作系统本身，因此 C 语言和类 UNIX 以及 Linux 具有很好的结合性。

相对而言，将 Windows 下编写带有 Windows 特性的 C 语言程序移植到类 UNIX 或 Linux 下，其工作量远大于类 UNIX 或 Linux 之间的相互移植的工作量，也大于将类 UNIX 或 Linux 下开发的 C 程序移植到 Windows 下的工作量。这里也建议那些一线的教育工作者，如同讲解汇编最好结合计算机组成原理一样，教授 C 语言的知识，最好结合类 UNIX 或 Linux 操作系统一并讲解。

如果你是一个尚未在类 UNIX 或 Linux 下开发的 C 语言初级程序员，则建议你应尽快熟悉并掌握某个类 UNIX 以及 Linux 的操作系统。例如，Mac OS X 就是稳定且具有良好应用界面的类 UNIX 系统。本书中的所有 C 语言程序，包括我近年的 C 语言工程开发均在一台 MacBook Pro 上完成。而基于 Linux 内核的操作系统选择也很多，例如，早年我曾在 Ubuntu 下进行 C 语言的开发。

建议 3：

除特定开发目标（如针对某特定硬件系统所设计的特定开发工具平台），正常的 C 语言

设计应在代码编辑器下编辑，在命令行下调用脚本、make 等工具开展工作，而非使用某种无目标系统特性的集成开发环境（Integrated Development Environment，IDE）。

集成开发环境主要包括编辑器、编译器、调试器和图形用户界面工具等。一些诸如 Sublime Text 等第三方编辑器的性能优良，比具有同样编辑功能的 IDE 更为轻巧。而长期使用某种 IDE，会逐步忽略了该环境对 C 代码组织上的特殊性（对于初学者甚至不了解这些特殊性），这会对以后调整 C 语言的开发环境有很多不利影响；采用断点、跟踪的调试器并不适用于连续运行下的各种情况跟踪（后续会在第 2 章展开讨论）；单纯的编译器对于 C 语言开发并不足够，这需要 make 和 shell 脚本等其他工具组成的工具链（后续会在第 3 章展开讨论）来提高你的工作效率；基于 C 语言的设计目标极少有针对图形应用界面的设计，因此图形界面的设计任务使用 C 语言开发并不适合。

我最初在 Turbo C 的 IDE 和 visio C++ 下学习 C 语言并设计程序，它们易于初学者上手，但“严重”阻碍初学者对 C 语言工程开发设计方法的掌握及应用。一种较为“偏激”的说法，如果你使用 IDE 写 C 程序，则你仅仅是在写 C 语言的程序，而不是在利用 C 语言按照工程化的组织方式开发一个系统。每个团队基于自身的业务背景、设计目标，会使用 C 语言、脚本等工具去构建和完善自身的工具包，组织成工具链，帮助自身提高 C 程序设计的效率。用 C 语言开发工具服务于 C 语言的开发，这是 C 语言程序员应当具备的能力。

建议 4：

除非你参与开源项目或希望你的设计目标以开源方式推广，否则更建议初级程序员使用“传统”的版本控制软件。此处“传统”指按照集中化管理的版本控制理论设计所开发出的版本控制软件，典型的如 CVS、SVN、Perforce 等。非“传统”的，如目前在各个开源社区中流行的 Git。此处并非说后者不好。一个产品，包括未来你所要设计的系统，“好”与“不好”都需要基于具体的应用场景来讨论。团队内高度协同的开发和基于开源社区（全球化）的开发，在分工组织模式上差异很大。后者极少出现两个程序员针对同一个 C 文件密集地进行修正调整（这需要以天或半天为单位，相互合并对方最新的代码），而在团队开发中，这种事情并不少见。我尊敬并赞赏那些为开源软件做出贡献的程序员，但作为初级程序员的你，我更建议你先在集群管控的团队下锻炼好自身的开发能力，再去学习开源系统的设计方法和使用面向开源软件开发的特有工具去参与开源软件的设计。

上述 4 条，仅仅是我个人的建议，既不是“标准”，也不是“守则”，与本书后续针对模块化设计所探讨的“规则”一样，它们只是建议，当然这些建议和规则有效帮助了我个人的开发工作，它们是否适合你，需要你自己的思考和实践。

1.1　C 的编译链接与文件引用

1.1.1　一个小程序

我不知道以下的程序是否算作最简的 C 语言程序，但它足够小，同时包含了很多初级程

试读结束：需要全本请在线购买： www.ertongbook.com

序员忽略的内容。代码如下：

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

上述程序存储为 C 文件前，我们先按照以下命令组织磁盘目录。

```
mkdir test
cd test
mkdir src
mkdir inc
mkdir obj
mkdir bin
```

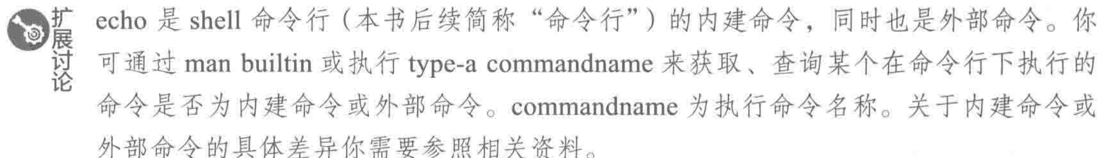
此时，当前目录为你刚才创建的 test 目录。其中，src 我们仅存储 C 文件，inc 则存储后续讨论到的头文件，obj 存储编译后的对象文件，bin 存储链接后形成的库或执行文件。这种组织方式并不是某种严格的规定，不按照这种组织方式，不代表不能构建 C 程序，但很多工程代码，采用了类似这样的组织方式，总是有一定理由的。

将上述三行语句，保存在当前目录下的 src/test.c 中，在当前目录下执行如下命令：

```
gcc -c src/test_main.c -o obj/test_main.o
gcc obj/test_main.o -o bin/test_main
bin/test_main
echo $?
bin/test_main 1 2
echo $?
bin/test_main 1 2 3
echo $?
```

上述第一行的命令为编译，你可以通过是否存在一个 -c 的选项来判断。第二行的命令为链接，它构建了可执行文件（gcc 通过缺少 -c 来判断）。第三行命令是执行生成在 bin 子目录（也可称为文件夹）下的执行程序 test_main。

echo $? 是用来检测最近一个执行操作的返回。随后是另两组再次执行与显示的操作。本书后续讨论中，若无特殊说明，则将第一行和第二行的两个操作，统一简称为“编译链接”，而第三行的操作，简称为“执行”。



echo 是 shell 命令行（本书后续简称“命令行”）的内建命令，同时也是外部命令。你可通过 man builtin 或执行 type-a commandname 来获取、查询某个在命令行下执行的命令是否为内建命令或外部命令。commandname 为执行命令名称。关于内建命令或外部命令的具体差异你需要参照相关资料。

上述命令执行后，应当分别返回 1、3、4。从上述三行的 C 语言程序中，你应该了解到，此时仅仅是返回了 main 函数的第一个参数，它表示当前命令执行时一共存在几个参数（包含

执行程序文件名本身）。

main 函数参照 C 语言国际标准的内容，它有两种形式，另一种如下：

```
int main(void);
```

但我建议不使用第二种。无论你所设计的应用程序是否需要跟随参数，保留它总没有错。而实际上大多数程序总需要一些给入参数，以方便程序在初始化时有一定选择性，哪怕你的程序的初始化参数均是通过文件读取，在 main 函数入口，给入一个参数文件存储位置的信息，这总比执行程序必须在特定目录下获取参数文件要人性化得多。

main 在 C 语言中是非常特殊的一个函数。对于链接形成的可执行程序，main 函数是整体程序的主入口。当然它存在于哪个 C 文件中并不重要。在 1.5 节的小模块举例中你会发现，更多的工作会从 main 函数中移除，而尽量保证 main 函数的简洁。main 函数里主要描述一个系统中（按照大类区分）各模块的配置及调度逻辑。而不要如学校交作业那样，一个作业内容，全部由一个 main 函数中实现。在后续章节，你会发现，一个模块的代码甚至不包括存放 main 函数的 test_XXX_main.c 文件，而后者仅是作为调用该模块进行测试的入口测试文件。这样做是为了方便一个工程的开发成果与其他工程整合利用。较为复杂的系统，更多情况下是切割成小块分别处理，而不是集中在一个工程中开发。

1.1.2 链接与文件引用

现在我们对上述 test_main.c 文件的内容做如下扩展：

```
#define MIN_PARAM_NUM 3
int chk_param(int argc,char *argv[]){
    if (argc < MIN_PARAM_NUM){
        return 0;
    }else{
        return 1;
    }
}
int main(int argc ,char *argv[]){
    return chk_param(argc,argv);
}
```

chk_param 函数不难理解，用于检测参数是否大于 3。编译链接后执行并使用 echo $? 观测结果。这里使用了“#define”宏操作将 3 这个数值通过一个名称为 MIN_PARAM_NUM 的标识符（简称“宏名称”）替代表示。

这种宏名称，在编译器进行编译前会通过预编译操作，将该标识符转换回文本内容 3。我们执行以下命令：

```
gcc src/test_main.c -E -P
```

此时会在屏幕上打印出经过预编译处理后的程序源码。预编译完成的工作比较有限，大

多数情况主要包括对宏的操作和引用文件的处理。上述“#define”的宏定义操作，会在后续章节继续展开讨论。本节先讨论引用文件。

我们对上述代码进行调整，将两个函数在 C 文件中的位置对调。此时编译，会出现类似如下的警告（实际是否出现，取决于你编译器的一些选项）。

```
warning: implicit declaration of function 'chk_param' is invalid ...
```

这里的大意是说，不明确在某一位置所调用函数“chk_param”的函数接口类型。

C 语言的编译器，是从 C 文件顶端依次向下读取内容并分析。对于出现调用一个函数的情况，需要编译器已知该函数的接口规则，否则编译器无法通过正确的汇编指令去组织函数调用中数据传递的操作。如果调用一个函数之前，C 文件中并没有出现过该函数接口的描述，编译器并不会停止工作，而是采用默认方式处理。如果默认方式恰巧符合你所设计的函数接口，就不会令程序出现错误，但这仅仅是碰巧的行为。

有时，你不可能将所有被调用的子函数写在调用者之前，如两个函数之间相互调用。因此合理的做法是对每个要被调用的函数，将其接口的情况（函数接口类型声明）在 C 文件最前端写出。组织方式如下：

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

int main(int argc ,char *argv[]){
    return chk_param(argc,argv);
}
#define MIN_PARAM_NUM 3
int chk_param(int argc,char *argv[]){
    ...
}
```

上述第一行语句是对 chk_param 函数接口类型的声明（以下简称“函数声明”）。编译器在对 main 函数进行编译时，已经了解了 chk_param 的接口情况，因此便可有针对性地组织数据传递的汇编指令。

一次编译器的执行，只会针对一个 C 文件，一个工程包括很多 C 文件，则需要编译器的多次执行，这将在第 3 章展开讨论。编译器并不在意被调用的函数是否存在于当前 C 文件中，链接时才会确定具体的函数入口位置。我们甚至可以如下组织源码，在 test_main.c 文件中，仅有如下代码：

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

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

其余代码则转移存储在 src/param.c 文件中，该文件的代码清单如下：

```
#define MIN_PARAM_NUM 3
int chk_param(int argc,char *argv[]){
    if (argc < MIN_PARAM_NUM){
        return 0;
    }else{
        return 1;
    }
}
```

执行以下命令：

```
gcc -c src/test_main.c -o obj/test_main.o
gcc -c src/param.c -o obj/param.o
gcc obj/param.o obj/test_main.o -o bin/test_main
```

上述第一、第二个命令是针对两个不同 C 文件的编译，而最后一个命令是将它们进行链接。如果你在执行了上述第一个命令后，便执行如下命令：

```
gcc obj/test_main.o -o bin/test_main
```

则会出现如下错误，它不是警告。

```
Undefined symbols for architecture XXXX:
"_chk_param", referenced from:
 _main in test_main.o
```

这个错误的提示是说，函数“_main”中，出现没有定义的符号“_chk_param”。这里的“_main”实际是 main 函数编译后形成的汇编函数。与 C 文件中的函数对应的汇编函数，后者的名称是前者名称增加前缀“_”。

链接器在链接时，需要对各个调用函数的指令设置具体的地址，否则机器执行时无法跳转到目标函数的有效起始位置。

C 语言的编译链接工作的详细原理和流程你可以参照相关资料做进一步的了解。

扩展讨论

这种以文件为单位进行编译形成对象文件，再汇聚各个编译后的对象文件进行链接形成执行程序的方式，实际对应了基础的模块化设计思想。

一个系统分解为多个模块，每个模块对应的实现代码通过不同的 C 文件或不同组的 C 文件进行组织，并分别独立开发、测试，最终再整合成系统。

回到代码中，在 test_main.c 中，包含了 param.c 中的函数接口声明。param.c 中的函数现在只被 test_main.c 中的 main 函数调用，但以后也可能被其他 C 文件中某个函数调用。如果在每个存在调用 chk_param 的 C 文件中都编写对 chk_param 函数声明的代码，实在过于烦琐。

一种合理的做法是使用 C 语言的 #include 预编译操作命令。它可以将后续的文件名所对应的内容，插入当前位置。通常所插入内容存储为 .h 后缀的文件（以下简称“头文件”）。

现在我们将上述函数声明的代码存储在一个名为 param.h 的文件中，该文件存储于 inc 目录下。而在 test_main.c 文件中使用如下操作：

```
#include "param.h"
```

这里的语句是告诉预编译器，将文件名为“ param.h”的所有文本内容从本行位置开始插入。此时对于 test_main.c 的编译，需要执行如下命令：

```
gcc -c -Iinc src/test_main.c -o obj/test_main.o
```

这里多了一个参数 -Iinc，它是头文件路径参数 -I 和头文件路径 inc 两个内容的合写。这样，编译器会尝试在 inc 中寻找 param.h。

#include 包括另一种文件描述方式，如下：

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

它使用 <> 而非 ""，这两者的不同，以及预编译过程中存在多个头文件路径时头文件的搜索规则，你可参照相关资料做进一步了解。

我们可以将需要共享给其他 C 文件的内容存放在头文件，并通过文件引用的方式，合并到对应的 C 文件里。而头文件内部，也可以去引用其他文件，这些并入的内容，最终都会插入到引用该头文件的 C 文件中，等待后续的预编译处理。

一个基础的概念，C 语言的编译链接并不针对头文件，而仅仅针对 C 文件。头文件的引用是编译前的预编译处理完成的工作，不要把头文件看作一个 C 程序的基础组成，而应当理解成头文件的内容是 C 程序中 C 文件的组成内容。

如下面章节的例子，我们在使用一个标准库函数 printf 时，需要引用一个 stdio.h 的文件。但 printf 函数并不在这个文件里，而是在标准库里。你引用 stdio.h 文件，是为了将 stdio.h 的整体内容作为你自己 C 文件的组成内容之一，其中包括了 printf 函数的接口类型声明。

对于上述的例子，可通过执行如下命令查看预编译对引用文件的处理结果：

```
gcc -Iinc src/test_main.c -E -P
```

1.2 函数、数据与作用域

1.2.1 全局函数与局部函数

1.1 节介绍了一个函数可被另一个 C 文件调用的方法。我们只需要将该函数接口声明放入一个头文件中，而调用者引用该文件，获取该函数的接口声明，便可以有效地调用。

如上讨论，编译仅仅针对一个 C 文件形成对象文件，而链接可以对给入的多个对象文件进行整合关联。在链接中，能够跨文件利用函数，其作用域是在本次链接所覆盖的整体范围，因此也称这些函数为全局函数。

广义的作用域，实际包含两个维度，其一是如上讨论的范围，称为狭义的作用域；其二指的是生命周期。在本章后续讨论中，如果独立地描述“作用域”，则指广义的作用域。

与全局函数对应的则是局部函数。它的作用域仅被局限在本 C 文件中。全局函数是默认的，而局部函数则需要在函数定义前增加 static 的关键词。由于全局函数作用域针对链接所覆盖的整体范围，因此不同 C 文件内的全局函数不能出现重名，否则链接时不能确定使用哪一个。相反，局部函数则可在不同 C 文件中存在相同的名称。如果你在头文件中定义了一个函数（不是函数接口声明），当两个 C 文件引用该头文件时，该函数定义内容同时会出现在两个 C 文件中，如果这个函数定义没有 static 关键词，它会被默认为全局函数，此时链接，则会出现重名错误。而如果定义为局部函数，则在两个文件中，存在两个作用域仅为自身 C 文件的局部函数，这并不会导致链接错误。

扩展讨论

全局函数、局部函数的差异在于作用域，这种差异使得工程应用上对它们的一些处理要求也存在差异。

一个函数，在处理外部给入的数值时，需要对数值是否在合理的范围进行检测，例如，你将使用到一个指针，如果它指向地址 0 时，你对该地址进行操作，会出现错误。

对给入数值进行范围检测的处理逻辑，我们简称为“边界检测”。没有边界检测的程序很难想象它处理数据的适用性。但每个函数都进行边界检测，既烦琐又降低了程序的执行效率。

由于 C 语言局部函数和全局函数的作用域不同，结合模块化设计的封装思想，在实际设计中可按如下规则确定一个函数是否需要进行边界检测。

所有全局函数需要对给入参数进行边界检测；而所有局部函数不需要对给入参数进行边界检测，其检测工作应由调用者完成。调用本 C 文件外的全局函数时，边界检测工作由被调用的全局函数完成，调用者不做检测。

一个 C 文件内部函数之间的调用关系，在设计该 C 文件代码时便可确定，调用者（函数）可明确知道被调用的局部函数对数值范围的要求。

而一个设计完成的 C 文件，其全局函数可能日后被很多其他 C 文件调用，你不能保证外部调用者一定能按照本 C 文件中内在处理逻辑的要求传入数据。因此它们总需要进行边界检测。

1.2.2　数据与数据的类别

在讨论数据前，我们首先针对“C 语言编程”这个背景域给出一个对“数据”的定义：具有用于存放具体数值的存储空间的待处理对象。

给出这个定义，是希望你能非常明确数值是数值，数据是数据。这对你理解诸如数组、指针等概念非常有帮助。

试读结束：需要全本请在线购买： www.ertongbook.com