高等学校“十三五”规划教材

C语言程序设计

实验指导与课程设计

邓达平　彭　洁　谢小云　编著

北　京
冶 金 工 业 出 版 社
2019

内 容 简 介

编程实践是学好C语言的“秘籍”。只有大量地进行程序设计实践，才能深刻地理解C语言的语法和技巧，进而成为“编程高手”。

本书是与《C语言程序设计》教材相配套的实验和课程设计指导书。全书共分为3篇。第1篇为理论基础篇，包括第1～3章，主要介绍C语言编程规范、编程环境Visual C++ 6.0及实验报告撰写规范。第2篇为核心实验篇，包括第4～13章，主要介绍C语言编程实验，并按照对应理论教材的章节顺序编排，可与教材同步使用。第3篇为综合应用篇，包括第14～16章，主要介绍C语言课程设计的相关内容；不仅介绍了课程设计大纲、EasyX，而且列举了一个课程设计实例。

本书内容全面，突出实践能力的培养，适合高等院校工科类专业学生使用，也可供广大C语言程序员参考。

图书在版编目(CIP)数据

C语言程序设计实验指导与课程设计/邓达平，彭洁，
谢小云编著. --北京：冶金工业出版社，2019.3
高等学校“十三五”规划教材
ISBN 978-7-5024-6173-7

Ⅰ.①C… Ⅱ.①邓… ②彭… ③谢… Ⅲ.①C语言—
程序设计—高等学校—教学参考资料 Ⅳ.①TP312.8

中国版本图书馆CIP数据核字(2019)第036026号

出 版 人 谭学余
地 址 北京市东城区嵩祝院北巷39号 邮编 100009 电话 (010)64027926
网 址 www.cnmip.com.cn 电子信箱 yjcbs@cnmip.com.cn
责任编辑 纵晓阳 路亚妮 美术编辑 易 帅 版式设计 刘 芬
责任校对 何立兵 责任印制 李玉山 张启敏
ISBN 978-7-5024-6173-7
冶金工业出版社出版发行；各地新华书店经销；三河市鑫鑫科达彩色印刷包装有限公司印刷
2019年3月第1版，2019年3月第1次印刷
787mm×1092mm 1/16；13.5印张；342千字；209页
39.80元
冶金工业出版社 投稿电话 (010)64027932 投稿信箱 tougao@cnmip.com.cn
冶金工业出版社营销中心 电话 (010)64044283 传真 (010)64027893
冶金工业出版社天猫旗舰店 yjgycbs.tmall.com

Preface

前　言

计算机程序设计是一门实践性非常强的课程，通过边学边练的方式，可以达到事半功倍的效果。因此，必须对编程实践给予足够的重视。编写程序最好的方法是在计算机上操作，可以利用集成开发环境等工具对代码进行检查、编译、运行和调试，采用即刻获得结果的方式来验证代码的正确性。这对于提高编程能力来说是非常直接且有效的。

本书是《C语言程序设计》教材的配套实验与课程设计指导书，两者配合使用。本书与类似的实验指导书的不同之处在于，它在内容上不仅涵盖了C语言所有知识点所对应的实验，还特别对编程规范、编程环境、实验报告撰写规范进行了介绍，最后一章还列举了一个课程设计实例，使读者可以由易到难、循序渐进，学会构建C语言综合应用程序的方法。

从内容上看，全书分为3篇：第1篇为理论基础篇，包括第1～3章，主要介绍C语言编程规范、编程环境Visual C++ 6.0及实验报告撰写规范；第2篇为核心实验篇，包括第4～13章，主要介绍C语言编程实验，并按照对应理论教材的章节顺序编排，可与教材同步使用；第3篇为综合应用篇，包括第14～16章，主要介绍C语言课程设计的相关内容，不仅介绍了课程设计大纲、EasyX，而且列举了一个课程设计实例。此外，本书所涉及的程序代码均在Visual C++ 6.0软件下进行编译，程序调试和运行过程简单易学。

本书编写分工为：第1、3、7、8、11～13章由谢小云编写，第2、10、14～16章由邓达平编写，第4～6、9章由彭洁编写。全书由邓达平统稿。

本书作者有多年从事C语言教学的经验，对于C语言的学习和运用都有自己独特的方法。这点在本书的各部分内容中都有相应的体现。

由于作者水平所限，书中难免有不足之处，期待读者批评指正，提出宝贵意见。

作　者

2019年1月

Contents
目　录

第 1 篇　理论基础篇

第 2 篇　核心实验篇

第 3 篇　综合应用篇

第 1 篇

理论基础篇

第1章　C语言编程规范

1.1　概述

计算机作为人类社会发展中出现的重要工具，极大地提升了人类社会发展的速度。计算机的运行需要人们预先编写程序。程序的规模随着其功能的复杂性而变得非常大。如果不采用规范的书写方法，则无法保证程序代码清晰、简洁、安全，以及程序的效率、可移植性和可测试性等性能要求。本章将介绍在程序设计中应当遵循的较为常用的编程规范。

1.1.1　什么是编程规范

编程规范是指在编写代码时应当遵循的各种原则和要素，用以降低代码错误出现的概率，提高代码的可读性、可维护性、可重用性，从而编写出高质量的代码。通常，编程规范包括原则、规则和建议。其中，原则是指导思想，规则是具体约定，建议是编程时出于优化代码需要而做的约定。

1.1.2　编程规范总体原则

编程规范主要是为了规范程序员编写代码的行为，形成统一的代码约定。总的来说，编程规范的原则有以下几个方面：

1. 简洁清晰

代码编写应当简洁清晰，避免过分使用编程技巧。如果程序员过分追求复杂的编程技巧，必然会降低代码的可读性，并容易出现错误。保持代码的简单化应成为软件开发的基本要求。

2. 便于阅读

代码是写给读者看的，计算机所执行的是经过编译、汇编之后的机器语言，计算机不需要去看程序员写的计算机语言代码。因此，编写出来的代码应便于阅读，便于人们相互之间交流、维护、测试等。只有易读、易维护的代码才具有长久的生命力。

3. 风格统一

软件开发团队在编写代码时，应使用同一个编程规范，采用统一的编程风格。因此，应对软件开发团队进行编程规范的培训，使每个成员都能够熟悉编程规范的细节。此外，如果要重构或修改其他风格的代码，可以根据现有代码的风格继续编写，也可以使用格式转换工具将其转换为团队所规定的编程风格。

1.2　基本编程规范

根据编程规范的使用情况，可将其分为基本编程规范和高级编程规范，前者适合初学程序开发的人员，后者适合具有一定编程经验的人员。限于篇幅，本书只介绍基本编程规范，读者可以自行参考相关资源学习高级编程规范。

1.2.1　文件结构

1.文件头

文件头用于说明文件名称、版权、版本等信息，以便读者了解文件的主要功能和属性。文件头的格式如下：

```
/*************************************************************
**************************************************************
* Copyright © 2018,开发单位名称
* All rights reserved.
* 文件名：
* 描述：对文件的功能、作用进行描述
* 作者：
* 主要函数及功能：
*************************************************************
**************************************************************
* 当前版本：
* 修改历史
* 1.修改时间：
*   修改人员：
*   修改记录：
* 2.修改时间：
*   修改人员：
*   修改记录：
* ……
*************************************************************
**************************************************************/
```

其中，“修改历史”部分应根据修改的情况，每修改一次增加一项说明。此处仅说明该文件中对哪些函数进行了修改，修改细节可以在后面的函数头中加以说明。

2.函数头

函数头放置在每个函数的前面，用以说明函数的主要功能、入口参数和返回值等信息。其结构如下：

```
/*************************************************************
**************************************************************
* 函数名：
* 描述：对函数的功能、作用进行描述
* 输入：入口参数
* 输出：返回值
* 调用函数：调用本函数的函数清单
* 被调用函数：被本函数调用的函数清单
```

```
*************************************
******************************
* 作者:
* 当前版本:
* 修改历史
* 1.修改时间:
*   修改人员:
*   修改记录:
* 2.修改时间:
*   修改人员:
*   修改记录:
* ……
*************************************
******************************/
```

其中,“修改历史”部分与文件头中的类似,用于列出该函数的修改信息,每修改一次都应增加一项,且必须给出具体的修改细节,以便代码阅读者可以清楚地知道函数的修改过程。

3. **头文件**

在C语言中,头文件可用来调用库功能。在很多场合,源代码不便(或不允许)向用户公布,只要向用户提供头文件和二进制的库即可。用户只需要按照头文件中的接口声明来调用库功能,而不必关心接口是如何实现的。编译器会从库中提取相应的代码。头文件能加强类型安全检查,如果某个接口被实现或被使用,但其方式与头文件中的声明不一致,则编译器就会指出错误。这一简单的规则能极大地减轻程序员调试、改错的负担。

(1)原则:

原则 1-1:头文件中只存放“声明”,而不存放“定义”。

头文件只放置模块或单元对外的声明,包括函数声明、宏定义、类型定义等。内部使用的函数声明不应放在头文件中,内部使用的宏、枚举定义、结构体定义也不应放在头文件中。此外,变量定义也不应放在头文件中,而应放在“.c”文件中。

原则 1-2:头文件应当简洁,不应过于复杂。

如果头文件过于复杂,使用“#include”包含过多的头文件,会导致编译时间过长,甚至会出现循环包含头文件的情况,将极大地降低代码编译的效率。

(2)规则:

规则 1-1:在头文件中,应使用条件编译 ifndef、define、endif 结构来防止头文件的重复引用。

规则 1-2:使用#include <filename.h>来引用标准库的头文件。

规则 1-3:使用#include "filename.h"来引用用户定义的非标准库的头文件。

可以通过下面给出的头文件 filename.h 的内容,来理解上述3条规则。

```
/* 文件头(省略)*/
#ifndef _FILENAME_H_          //采用条件编译防止重复引用
#define _FILENAME_H_
```

```
#include <stdio.h>          //引用标准库的头文件
……
#include "myfile.h"         //引用用户定义的非标准库的头文件
……
void function1(……)          //声明全局函数
……
typedef struct {            //定义全局结构体类型
……
}newtype;

#endif                      //结束条件编译
```

规则 1-4：每一个“.c”文件都应有一个“.h”文件与之对应，用于声明该“.c”文件需要对外公开的接口。

规则 1-5：无论是在“.c”文件还是“.h”文件中，都不应引用用不到的头文件。

规则 1-6：禁止在头文件中定义变量。

如果在头文件中定义变量，那么这个变量可能会因为该头文件被其他“.c”文件引用，进而出现变量重复定义的情况。

规则 1-7：只能通过引用头文件的方式来使用其他“.c”文件提供的接口，禁止在“.c”文件中通过 extern 的方式使用外部的函数接口或变量。

(3)建议：

建议 1-1：应将属于同一个模块的多个“.c”文件存放到同一个目录下，并使用模块名命名该目录。编写一个以模块名命名的“.h”文件，作为这个模块对外的接口。

建议 1-2：如果一个模块由多个子模块组成，则为每个子模块编写一个“.h”文件，用以对外提供接口。

4.实例

下面通过一个简单的实例来了解程序的文件结构。假如某程序由一个结构体定义、一个 Display 函数和一个 main 函数构成，这些内容都在同一个“.c”源文件中。

```
#include <stdio.h>
typedef struct{                 //结构体定义
  int SNo;
  char SName[21];
}Student;
void Display(Student s)
{
  printf("学号：%d\n姓名：%s\n",s.SNo,s.SName);
}
void main()
{
  Student t={1001,"张三丰"};
```

```
    Display(t);
}
```

这种安排方式在简单程序中是可以的。但如果程序由几百个函数组成，甚至其中又分为若干模块，那么将所有的代码、定义都写在同一个文件中会显得非常凌乱，且不便维护和修改。这时可以考虑适当分解，将定义写在一个头文件（如 Definition.h）中，将函数声明（注意是声明，不是函数实现的定义）写在另一个头文件（如 Declaration.h）中，再将这些函数的实现写在一个源文件（如 Implementation.c）中，得到如下的文件结构及内容：

```
//Definition.h 头文件，结构体的定义
#include <stdio.h>
#ifndef_DEFINITION_
  #define_DEFINITION_
  typedef struct{              //结构体定义
    int SNo;
    char SName[21];
  }Student;
#endif
//Declaration.h 头文件，各种函数原型的声明
#include <stdio.h>
#include "Definition.h"       //包含结构体定义的头文件
#ifndef _MODULAR_
  #define_MODULAR_
  void Display(Student s);
#endif
//Implementation.c 源文件，各种函数的具体实现
#include "Definition.h"       //包含结构体定义的头文件
#include "Declaration.h"      //包含函数原型声明的头文件
void Display(Student s)
{
  printf("学号：%d\n 姓名：%s\n",s.SNo,s.SName);
}
//StuManager.c 源文件，main 函数所在的主源文件
#include <stdio.h>
#include "Definition.h"       //包含结构体定义的头文件
#include "Declaration.h"      //包含函数原型声明的头文件
                              //不需要包含函数实现的源文件
void main()
{
  Student t={1001,"张三丰"};
```

```
    Display(t);
}
```

在上述安排方式中，利用了C语言中的各种预编译指令#include、#define、#ifndef、#endif等，可以使大、中型程序的结构清晰，容易维护，也适合多人合作开发。

1.2.2　排版规范

代码的排版虽然不会影响代码的功能，但会影响可读性，也就带来了发生错误的可能性。代码的排版应美观、清晰、一目了然。

1. 规则

规则 2-1：程序块应使用缩进风格，并使用一个Tab键（Visual C++ 6.0中默认为4个空格）作为一次缩进的单位，或者每级使用空格缩进。

如果开发工具能够自动生成缩进，则可按照开发工具的规则。宏定义、条件编译语句等可以顶格书写。

① 不符合规范的写法：

```
if(a > b)
{
a++;
c=a;
}
else
{
b++;
c=b;
}
```

② 符合规范的写法：

```
if(a > b)
{
  a++;
  c=a;
}
else
{
  b++;
  c=b;
}
```

规则 2-2：一行只写一条语句，不允许将多条语句写到一行内。

① 不符合规范的写法：

```
int k=1;float j=2.1;
```

② 符合规范的写法：

```
int k=1;
float j=2.1;
```

规则 2-3：if、for、do、while、switch、case、default 等指令应独占一行，它们的执行语句应当缩进一级。无论它们的执行语句只有一行还是多行，都应使用成对的“{ }”，且“{”和“}”分别独占一行，并与 if、for、do、while、switch、case、default 等对齐。

```
if(count > 10)
{                       //独占一行，且与 if 对齐
  count++;              //即使只有一行，也应使用“{”和“}”，且应缩进一级
}                       //独占一行，且与 if 对齐
```

规则 2-4：当两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的操作符（如“－－”“＋＋”等），后面不应加空格。

① 逗号、分号只在后面加空格。

```
int a, b, c;
```

② 比较操作符，赋值操作符“＝”“＋＝”，算术操作符“＋”“％”，逻辑操作符“&&”“||”，位域操作符“<<”“^”等双目操作符的前后都加空格。

```
if(count >= 100)
a = b - c;
a /= 2;
a = c ^ 2;
```

③ “!”“～”“＋＋”“－－”等单目操作符前后均不加空格。

```
flag=!is_true;          //非操作“!”与内容之间
k++;                    // “++”“－－”与内容之间
```

④ “→”“.”前后都不加空格。

```
p→id=pid;               //“→”前后不加空格
```

⑤ 应将修饰符“*”和“&”紧靠变量名。

```
char *pname;
int *m, n;              //m 为 int 型指针，n 应为 int 型变量
```

采用本规则的目的在于使符号之间的界限清晰明了、便于阅读。但对于本身就比较清晰的语句，相关符号之间则不需要留空格，如括号的内侧就不需要加空格，多重括号之间也不必加空格。

2. 建议

建议 2-1：“//”及“/* */”注释符与所注释内容之间要用一个空格进行分隔。

采用这一方式，可以让注释信息更加清晰、方便阅读。

建议 2-2：源程序中关系较为紧密的代码应尽可能相邻。

该建议通常用于实现某一功能的多个指令属于关系较为紧密的代码。此外，同一类代码也可以作为相邻代码处理，如变量定义、变量初始化等。

1.2.3　注释规范

注释可以帮助读者理解代码，适当的注释是高质量代码的必需内容。

原则 3-1：优秀的代码不是通过注释来说明，而是让代码自解释。

人们可以通过将函数、过程、变量和结构体等进行有含义的命名，以及合理组织代码的结构，来实现代码的自解释。

原则 3-2：注释的内容应清楚明了、含义准确，避免出现二义性。

原则 3-3：注释不是对代码的重复描述，应从功能、作用等角度进行注释。

对于代码能够自解释的部分，不需要进行注释。注释的目的在于解释代码的目的、功能和采用的方法，提供代码之外的信息，帮助读者理解代码。因此，对于较难理解的代码应加以注释。

① 多余的注释：

```
i＋＝1;              //i 加 1
if(flag)             //如果 flag 为真
```

② 合适的注释：

```
int count;           //count 用于保存数据采集的数量
```

1.2.4　标识符命名规范

1. 标识符命名的常见风格

目前，较为常见的标识符命名风格有以下几种：

(1)UNIX like 风格。

在 UNIX like 风格中，标识符使用小写字母，每个单词直接用下画线进行分割，如 text_mutex、kernel_text_length。

(2)Windows like 风格。

在 Windows like 风格中，标识符使用大小写字母混用的方式，每个单词直接连接在一起，每个单词的首字母大写，如 TextMutex、KernelTextLength。

(3)匈牙利命名法。

匈牙利命名法的主要思想是在变量或函数名中输入前缀，以便它们能够更易于被理解。在这种风格中，标识符包括 3 个部分：基本类型、一个或多个前缀、一个限定词，如字符变量以 ch 为前缀，指针变量以 p 为前缀。匈牙利命名法会让标识符的命名和使用变得烦琐。

实际上，每一种命名风格都存在优缺点，无法使所有程序员达成统一的意见。因此，对于标识符命名的规则，就开发团队来说，团队内部统一标准即可；对于程序员来说，则应确定一种风格，并一直遵循这一风格，且不能随意改变。

2. 原则

原则 4-1:标识符的命名应使用具有与其功能相对应的、有明确含义的单词,或者是通用的缩写,避免使人产生误解。

① 不符合规范的写法:

```
int a,b;              //使用了随意的字符
int n_comp_conns;     //使用了模糊的缩写
```

② 符合规范的写法:

```
int error_number;
int number_of_completed_connection;
```

原则 4-2:仅使用通用的缩写,不应使用汉语拼音。

一般单词的缩写方法:较短的单词,可通过去掉元音形成缩写,如 count 可缩写为 cnt,clock 可缩写为 clk 等。对于较长的单词,则可取单词的头几个字符形成缩写,如 argument 可缩写为 arg,increment 可缩写为 inc 等。

原则 4-3:命名规范应与所使用系统风格一致,同一个项目中风格应统一,开发团队内部应保持统一的命名风格;程序员个人应坚持使用自己所确定的风格,除非项目开发团队要求改变风格,以便达到团队内部统一。

1.2.5 程序的可读性

1. 规则

规则 5-1:注意运算符的优先级,并用括号明确表达式的操作顺序,避免使用默认优先级。

代码主要是供读者阅读的,因此采用嵌套的括号可以防止阅读程序时产生误解,防止因默认的优先级与设计思想不符而导致程序出错。

对于以下指令:

```
word=(high << 8) | low        (1)
if((a | b) && (a & c))        (2)
if((a | b) < (c & d))         (3)
```

如果书写为:

```
word=high << 8 | low
if(a | b && a & c)
if(a | b < c & d)
```

由于“high<<8 | low”是“(high<<8)| low”,“a | b && a & c”是“(a | b)&&(a & c)”,(1)、(2)虽然不会出错,但语句不易理解。“a | b<c & d”其实是“a |(b<c)& d”,(3)造成了判断条件出错。

规则 5-2:避免使用不易理解的数字,可用有意义的标识符来代替。涉及物理状态或者含有物理意义的常量,不应直接使用数字,必须用有意义的枚举或宏来代替。

2. 建议

建议 5-1：源程序中关系较为紧密的代码应尽可能相邻。

建议 5-2：不要使用难以理解、技巧性很高的语句，除非很有必要。

使用高技巧编写的语句虽然看上去显得比较简洁，但是高技巧语句不等于高效率的程序。实际上，程序的效率关键在于算法。

对于下述表达式，考虑不周就可能出问题，也较难理解。

```
 * stat_poi +++=1;
 * ++stat_poi+=1;
```

应分别修改如下：

```
 *stat_poi+=1;
stat_poi++;          //这两条语句的功能相当于" * stat_poi +++=1;"
++stat_poi;
 *stat_poi+=1;       //这两条语句的功能相当于" * ++stat_poi+=1;"
```

1.2.6　变量和结构体的规范

1. 原则

原则 6-1：变量应当具有单一的功能，不应让其具有多种用途。

一个变量仅用来表示一个特定功能，不能赋予一个变量多种用途，即同一变量取值不同时，其代表的意义也不同。

原则 6-2：结构体功能单一，不要设计面面俱到的数据结构。

相关的一组信息才是构成一个结构体的基础，结构体的定义应该明确地描述一个对象，而不是一组相关性不强的数据集合。

设计结构体时应力争使结构体代表一种现实事物的抽象，而不是同时代表多种。结构体中的各元素应代表同一事物的不同侧面，而不应把没有关系或关系很弱的不同事物的元素放到同一结构体中。

原则 6-3：不用或者少用全局变量。

单个文件内部可以使用 static 的全局变量。全局变量应该是模块的私有数据，不能作为对外的接口使用，使用 static 类型定义，可以有效防止外部文件的非正常访问。建议定义一个 STATIC 宏，在调试阶段，将 STATIC 定义为 static，版本发布时改为空，便于后续的打补丁等操作。

2. 规则

规则 6-1：去掉没必要的公共变量。

公共变量是增大模块间耦合的原因之一，故应减少没必要的公共变量，以降低模块间的耦合度。

规则 6-2：仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。

在对变量声明的同时，应对其含义、作用及取值范围进行注释说明。同时，若有必要，还应说明它与其他变量的关系。

规则 6-3：明确公共变量与操作此公共变量的函数或过程的关系，如访问、修改及创建等。

明确过程操作变量的关系后，将有利于程序的进一步优化、单元测试、系统联调及代码维护等。这种关系的说明可在注释或文档中描述。

规则 6-4：当向公共变量传递数据时，应十分小心，防止赋予不合理的值或越界等现象发生。

对公共变量赋值时，若有必要，应进行合法性检查，以提高代码的可靠性、稳定性。

规则 6-5：防止局部变量与公共变量同名。

若使用了较好的命名规则，则此问题可自动消除。

规则 6-6：严禁使用未经初始化的变量作为右值。

在 C 语言程序中引用未经赋值的指针，经常会引起系统崩溃。

1.2.7 函数和过程的规范

1. 原则

原则 7-1：对所调用函数的错误返回码要仔细、全面地处理。

原则 7-2：明确函数功能，精确（而不是近似）地实现函数设计。

原则 7-3：在同一项目组应明确规定，对接口函数参数的合法性检查应由函数的调用者负责还是由接口函数本身负责，默认是由函数调用者负责。

对于模块间接口函数参数的合法性检查问题，往往有两种极端现象：一种是调用者和被调用者对参数均不做合法性检查，结果就遗漏了合法性检查这一必要的处理过程，从而产生问题隐患；另一种是调用者和被调用者均对参数进行合法性检查，这种情况虽不会造成问题，但产生了冗余代码，降低了效率。

2. 规则

规则 7-1：防止将函数的参数作为工作变量。

将函数参数作为工作变量，可能会错误地改变参数内容，所以很危险。对于必须改变的参数，最好先用局部变量代之，最后将该局部变量的内容赋给该参数。

① 不符合规范的写法：

```
void sum_data(unsigned int num,int *data,int *sum)
{
  unsigned int count;
  *sum=0;
  for(count=0;count<num;count++)
  {
    *sum +=data[count];          //sum 成了工作变量，不合适
  }
}
```

② 符合规范的写法：

```
void sum_data(unsigned int num,int *data,int *sum)
{
  unsigned int count;
  int sum_temp;                  //增加局部变量作为工作变量
```

```
    sum_temp=0;
    for(count=0;count<num;count ++)
    {
      sum_temp +=data[count];
    }
    * sum=sum_temp;                    //最后将局部变量的值赋给实际的参数
}
```

规则 7-2：不要设计多用途的函数，一个函数仅完成一个功能即可。函数的规模尽量限制在 200 行以内。

规则 7-3：为简单功能编写函数。

虽然为仅用一两行代码就可完成的功能编写函数好像没有必要，但用函数可使功能明确化，不仅增加程序的可读性，而且方便程序的维护与测试。

如下语句的功能不是很明显：

```
value=(a > b) ? a:b;
```

做如下修改就很清晰了：

```
int max(int a,int b)
{
  return((a > b)? a:b);
}
value=max(a,b);
```

或修改如下：

```
#define MAX(a,b)(((a)>(b))? (a):(b))
value=MAX(a,b);
```

规则 7-4：检查函数所有参数输入的有效性，检查函数所有非参数输入的有效性，如数据文件、公共变量等。

函数的输入主要有两种：一种是参数输入；另一种是全局变量、数据文件的输入，即非参数输入。函数在使用输入之前，应进行必要的检查。

规则 7-5：函数名应准确描述函数的功能，可使用动宾词组为执行某操作的函数命名。

一般可以参照如下方式进行函数命名：

```
void print_record(unsigned int rec_ind);
int input_record(void);
unsigned char get_current_color(void);
```

规则 7-6：避免函数中出现不必要的语句，防止程序中的垃圾代码。

程序中的垃圾代码不仅占用额外的空间，而且还常常影响程序的功能与性能，很可能给程序的测试、维护等造成不必要的麻烦。

规则 7-7：功能不明确、较小的函数，特别是仅有一个上级函数调用它时，应考虑把它合并到上级函数中，而不必单独存在。

模块中函数划分得过多，一般会使函数间的接口变得复杂。所以过小的函数，特别是扇入很低或功能不明确的函数，不值得单独存在。

规则 7-8：如果多段代码重复做同一件事情，那么在函数的划分上可能存在问题。

若此段代码各语句之间有实质性关联并且是完成同一功能的，则可以考虑把此段代码构造成一个新的函数。

规则 7-9：改进模块中函数的结构，降低函数间的耦合度，并提高函数的独立性及代码的可读性、效率和可维护性。优化函数结构时，应遵循以下原则：

(1)不能影响模块功能的实现。

(2)仔细考查模块或函数出错处理及模块的性能要求并进行完善。

(3)通过分解或合并函数来改进软件的结构。

(4)考查函数的规模，过大的函数要进行分解。

(5)降低函数间接口的复杂度。

(6)不同层次的函数调用要有较合理的扇入、扇出。

(7)函数功能应可预测。

(8)提高函数内聚(单一功能的函数内聚最高)。

规则 7-10：当一个过程(函数)中对较长变量(一般是结构体的成员)有较多引用时，可以用一个意义相当的宏代替。