第9章 软件实现

- 程序设计语言
- 程序设计风格
- 编码规范
- 程序效率与性能分析

9.1 程序设计语言

- 程序设计语言的性能
  - **从软件心理学及软件工程角度对程序设计语言的性能进行讨论。**

9.1 程序设计语言

- 软件心理学的观点

(1) 一致性。表示一种语言所使用符号的兼容程度、允许随意规定限制，以及允许对语法或语义破例的程度。

(2) 二义性。虽然语言的编译程序总是以一种机械的规则来解释语句，但读者则可能用不同的方式来理解语句。

(3) 简洁性。程序设计语言的简洁性用来表示为了用该语言编写程序，必须记忆的关于代码的信息量。

9.1 程序设计语言

- 软件心理学的观点

(4) 局部性。是指程序设计语言的综合特性。在编码的过程中，由语句组合成模块，由模块组装为程序系统结构，并在组装过程中实现模块的高内聚和低耦合，可使程序的局部性加强。

(5) 传统性。人们学习一种新的程序设计语言的能力受到传统的影响。

9.1 程序设计语言

- 软件工程的观点

(1) 详细设计应能直接地容易地翻译成代码程序。

(2) 源程序应具有可移植性。

(3) 编译程序应具有较高的效率。

(4) 尽可能应用代码生成的自动工具。

(5) 可维护性。

9.1 程序设计语言

- 程序设计语言的分类

➢ 从软件工程的角度，根据程序设计语言发展的历程，可以将程序设计语言大致分为4类。

(1) 从属于机器的语言——第一代语言

(2) 汇编语言——第二代语言

(3) 高级程序设计语言——第三代语言

从20世纪50年代就开始出现，它们的特点是用途广泛。典型的高级程序设计语言有ALGOL、FORTRAN、COBOL、BASIC、PASCAL、C、C ++、Lisp、PROLOG、Ada等。

9.1 程序设计语言

- 程序设计语言的分类

(4) 第四代语言（4GL）

- **4GL以数据库管理系统所提供的功能为核心，进一步构造了开发高层软件系统的开发环境，如报表生成系统、多窗口表格设计系统、菜单生成系统等。**
- **4GL提供了功能强大的非过程化问题定义手段，用户只需告诉系统做什么，而无须说明怎么做。**

9.1 程序设计语言

- 程序设计语言的选择

➢ **在选择编程语言时，可以考虑以下因素。**

(1) **应用领域：目标系统的应用领域不同，需要采取的系统开发范型也不同，所以要考虑支持相应范型的编程语言。**

(2) **系统用户的要求。**

(3) **编程语言自身的功能。**

(4) **编码和维护成本及开发环境。**

(5) **编程人员的技能。**

(6) **软件可移植性。**

9.2 程序设计风格

- 在软件生存期中，人们经常要阅读程序。特别是在软件测试阶段和维护阶段，编写程序的人和参与测试、维护的人都要阅读程序。
- 在编写程序时多花些工夫，讲求程序的风格，这将大量地减少人们读程序的时间。
- 本节对程序设计风格的4个方面，即**源程序文档化**、**数据说明的方法**、**语句结构**和**输入/输出方法**中值得注意的问题进行概要的讨论

9.2 程序设计风格

- 源程序文档化
  - **源程序文档化包括标识符的命名、安排注释以及程序的视觉组织等。**

9.2 程序设计风格

- 标识符的命名

➢ **标识符包括模块名、变量名、常量名、标号名、子程序名以及数据区名、缓冲区名等。这些名字应能反映它所代表的实际东西，使其能够见名知意，有助于对程序功能的理解。**

➢ **应当选择精练的意义明确的名字，才能简化程序语句，易于对程序功能的理解。**

9.2 程序设计风格

- 程序的注释
  - **正确的注释能够帮助读者理解程序，为测试和维护阶段提供明确的指导。**
  - **注释行的数量占到整个源程序的1/3到1/2。**
  - **注释分为序言性注释和功能性注释。**
  - **序言性注释通常置于每个程序模块的开头部分，它应当给出程序的整体说明，对于理解程序本身具有引导作用。**

9.2 程序设计风格

- 序言性注释

➢ **有些软件开发部门对序言性注释做了明确而严格的规定，要求程序编制者逐项列出。**

(1) 程序标题。

(2) 有关本模块功能和目的的说明。

(3) 主要算法。

(4) 接口说明：包括调用形式，参数描述，子程序清单。

(5) 有关数据描述：重要的变量及其用途，约束或限制条件，以及其他有关信息。

(6) 模块位置：在哪一个源文件中，或隶属于哪一个软件包。

(7) 开发简历：模块设计者，复审者，复审日期，修改日期、有关说明等。

9.2 程序设计风格

- 功能性注释
  - **功能性注释嵌在源程序体中，用以描述其后的语句或程序段，也就是解释下面要“做什么”，或是执行了下面的语句会怎么样。**
  - **例如，下面的注释行仅仅重复了后面的语句，对于理解它的工作并没有什么作用。**

```
/* Add amount to total */
total = amount + total;
```

如果注明把月销售额计入年度总额，便使读者理解了下面语句的意图：

```
/* Add monthly-sales to annual-total */
  total = amount + total;
```

9.2 程序设计风格

- 功能性注释

➢ **书写功能性注释，要注意以下几点。**

(1)用于描述一段程序，而不是每一个语句。

(2)用缩进和空行，使程序与注释容易区别。

(3)注释要正确。

9.2 程序设计风格

- 视觉组织—空格、空行和移行

➢ **空格**：恰当地利用空格，可以突出运算的优先性，避免发生运算的错误。例如，将表达式

```
(a < -17)&&!(b <=49)||c
```

写成

```
(a < -17) && !(b <=49) || c
```

就更清楚。

➢ **空行**：自然的程序段之间可用空行隔开。

9.2 程序设计风格

- 视觉组织—空格、空行和移行
  - **移行**：**移行也叫做向右缩格。对于选择语句和循环语句，把其中的程序段语句向右做阶梯式移行，可使程序的逻辑结构更加清晰，层次更加分明。**

```
if （…）
        if （…）
        {
                ……
        }
        else
        {
                ……
        }
else
{
        ……
}
```

9.2 程序设计风格

- 数据说明标准化
  - 为了使程序中数据说明更易于理解和维护，在编写程序时，需要注意数据说明的风格。

(1) 数据说明的次序应当规范化，使数据属性容易查找，也有利于测试、排错和维护。

① 常量说明。

② 简单变量类型说明。

③ 数组说明。

④ 公用数据块说明。

⑤ 所有的文件说明。

9.2 程序设计风格

- 数据说明标准化

(2) 当多个变量名用一个语句说明时，应当对这些变量按字母顺序排列。例如，将

int size，length，width，cost，price;

进行如下改写更好。

int cost，length，price，size，width;

(3) 对于复杂的数据结构，应当使用注释对其进行说明。

9.2 程序设计风格

- 语句结构简单化
  - **在编码阶段，语句结构要力求简单、直接，不能为了片面追求效率而使语句复杂化。**

(1) 在一行内只写一条语句，并且采取适当的移行格式，使程序的逻辑和功能变得更加明确。例如，下面程序的可读性很差。

```
for (i=1; i<=n-1; i++) { t=i; for (j=i+1; j<=n; j++) if (a[j]<a[t]) t=j; if (t!=i) {temp=a[t]; a[t]=a[i]; a[i]=temp}}
```

9.2 程序设计风格

- **可以将上面的程序段改写成如下形式：**

```
for (i=1; i<=n-1; i++)
{
    t=i;
    for (j=i+1; j<=n; j++)
        if (a[j]<a[t])
            t=j;
    if (t!=i)
    {
        temp=a[t];
        a[t]=a[i];
        a[i]=temp;
    }
}
```

9.2 程序设计风格

(2)程序编写首先应当考虑清晰性，不要刻意追求技巧性，使程序编写得过于紧凑。例如，有一个用C语句写出的程序段：

```
a[i] = a[i] + a[t];
a[t] = a[i] - a[t];
a[i] = a[i] - a[t];
```

如果改写成下面的程序段，其功能就能够一目了然了。

```
temp = a[t];
a[t] = a[i];
a[i] = temp;
```

9.2 程序设计风格

(3)程序编写得要简单，写清楚，直截了当地说明程序员的用意。 例如，下面是是一个有双重循环的程序段，得到的结果是一个N×N的二维数组。

```
for ( i = 1; i <= n; i++ )
    for ( j = 1; j <= n; j++ )
        v[i][j] = ( i / j ) * ( j / i )
```

如果写成以下的形式，就能让读者直接了解程序编写者的意图了。

```
for ( i=1; i <= n; i++ )
    for ( j=1; j <= n; j++ )
        if ( i == j )
            v[i][j] = 1.0;
        else
            v[i][j] = 0.0;
```

9.2 程序设计风格

(4)除非对效率有特殊的要求，程序编写要做到清晰第一，效率第二。

(5)避免使用临时变量而使可读性下降。例如，由于简单变量的运算比下标变量的运算要快，有的程序员为了追求效率，会将语句

```
x = a[i] + 1/a[i] ;
```

写成

```
ai = a[i];
x = ai + 1/ai;
```

这样做，虽然效率要高一些，但引进了临时变量，把一个计算公式拆成了几行，增加了理解的难度。

9.2 程序设计风格

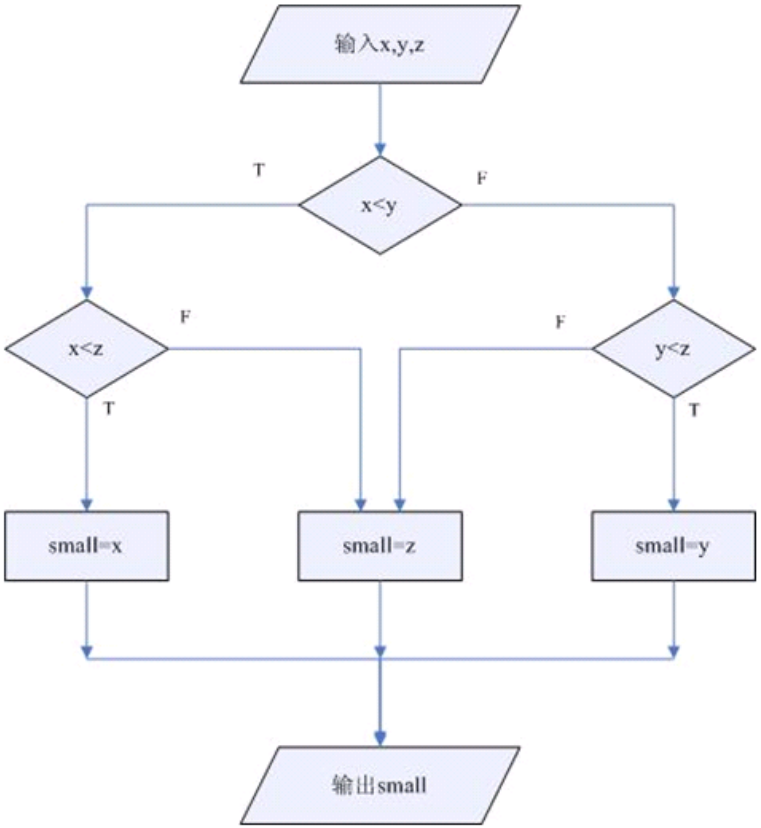
(6) 让编译程序做简单的优化。

(7) 尽可能使用库函数。

(8) 避免不必要的转移，同时如果能保持程序可读性，则不必用GOTO语句。

9.2 程序设计风格

- 例如，有一个求3个数中最小值的程序，对应的流程图及程序如下：



求 x、y、z 中最小者

```
        int x, y, z, small;
        scanf("%d %d %d", &x, &y, &z);
        if (x<y)    goto b30;
        if (y<z)    goto b50;
        small=z;
        goto b70;
b30:
        if (x<z)  goto b60;
        small=z;
        goto b70;
b50:
        small=y;
        goto b70;
b60:
        small=x
b70:
        printf("small=%d\n", small);
```

9.2 程序设计风格

(9)尽量只采用3种基本的控制结构来编写程序。

(10)避免使用空的else语句和if…then if…语句。

这种结构容易使读者产生误解。

例如，下面的程序可能产生二义性问题。

```
if ( char >= 'a' )
if ( char <= 'z' )
   cout <<  "This is a letter。";
 else
   cout << "This is not a letter。";
```

9.2 程序设计风格

(11) 避免采用过于复杂的条件测试。

(12) 尽量减少使用“否定”条件的条件语句。例如，如果在程序中出现

```
if ( ! ( char < '0' || char > '9' ) )
    ......
```

将其改成下面的语句，含义会更直接。

```
if ( char >= '0' && char <= '9' )
    ......
```

9.2 程序设计风格

- 输入/输出规范化
  - 输入/输出信息是与用户的使用直接相关的。输入/输出的方式和格式应当尽可能方便用户的使用。
  - 输入/输出的风格随着人工干预程度的不同而有所不同。

9.2 程序设计风格

- 输入/输出的原则

(1) 对所有的输入数据都进行检验，从而识别错误的输入，以保证每个数据的有效性。

(2) 检查输入项的各种重要组合的合理性，必要时报告输入状态信息。

(3) 使得输入的步骤和操作尽可能简单，并保持简单的输入格式。

(4) 输入数据时，应允许使用自由格式输入。

(5) 应允许默认值。

(6) 输入一批数据时，最好使用输入结束标志，而不要由用户指定输入数据数目。

9.2 程序设计风格

- 输入/输出的原则(续)

(7)在以交互式输入/输出方式进行输入时，要在屏幕上使用提示符明确提示交互输入的请求，指明可使用选择项的种类和取值范围。同时，在数据输入的过程中和输入结束时，也要在屏幕上给出状态信息。

(8)当程序设计语言对输入/输出格式有严格要求时，应保持输入格式与输入语句要求的一致性。

(9)给所有的输出加注解，并设计输出报表格式。

9.3 编码规范

- 在参考微软、Bell等公司编码规范的基础上，本节以C/C++为示例对编码规范给出简要介绍。
- 规范涉及版面、注释、标识符命名、变量使用、代码可测性、程序效率、质量保证、代码编译、单元测试、程序版本与维护。

1. 版面

(1) 程序块要采用缩进风格编写，缩进的空格数为4个。但对于由开发工具自动生成的代码可以有不一致。

(2) 相对独立的程序块之间、变量说明之后应加空行。

(3) 较长的语句（ >80字符）要分成多行写，长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。

(4) 循环、判断等语句中的条件测试若有较长的表达式，则要进行适应的划分，长表达式要在低优先级操作符处划分新行，操作符放在新行之首。

1. 版面

(5) 不允许把多个短语句写在一行中，即一行只写一条语句。

(6) if、while、for、default、do等语句自占一行。

(7) 只用空格键，不要使用TAB键。以免用不同的编辑器阅读程序时，因TAB键所设置的空格数目不同而造成程序布局不整齐。

(8) 函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格，case语句下的情况处理语句也要遵从语句缩进要求。

1. 版面

(9) 程序块的分界符（如C / C++语言的大括号 ‘{’ 和 ‘}’）应各独占一行并且位于同一列，同时与引用它们的语句左对齐。

在函数体开始、类定义、结构定义、枚举定义以及if、for、do、while、switch、case语句中的程序都要采用如上的缩进方式。

1. 版面

- **例如，下面的程序段不符合规范。**

```
for ( ... ) {
... // 程序代码
}
if ( ... ) {
... // 程序代码
}
```

- **应书写为以下格式：**

```
for ( ... )
{
    ... // 程序代码
}
if ( ... )
{
    ... // 程序代码
}
```

1. 版面

(10) 在两个以上的变量、常量间进行判等操作时，操作符之前、之后或者前后要加空格；进行非判等操作时，如果是关系密切的操作符（如->、::），后面不应加空格。

由于留空格所产生的清晰性是相对的，所以，在已非常清晰的语句中没有必要再留空格，如括号内侧（左括号后面和右括号前面）不要加空格，多重括号间不必加空格。

2. 注释

- 注释的原则是有助于对程序的阅读理解。

(1) 一般情况下，注释量一般控制在20%到50%之间。

(2) 说明性文件（如头文件.h文件、.inc文件、编译说明文件.cfg等）头部应进行注释，注释必须列出：版权说明、版本、生成日期、作者、内容、功能、与其他文件的关系、修改日志等，头文件的注释中还应有函数功能简要说明。

(3) 源文件头部应进行注释，列出：版权说明、版本号、生成日期、作者、模块目的/功能、主要函数及其功能、修改日志等。

2. 注释

(4) 函数头部应进行注释，列出：函数的目的/功能、输入参数、输出参数、返回值、调用关系（函数、表）等。

(5) 注释的内容要清楚、明了、含义准确，防止注释二义性。

(6) 避免在注释中使用缩写，特别是非常用缩写。在使用缩写时或之前，应对缩写进行必要说明。

(7) 代码的注释应放在代码的上方或右方（对单条语句的注释）相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开。

2. 注释

(8) 对于所有有物理含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加注释，说明其物理含义。变量、常量、宏的注释放在其上方相邻位置或右方。

(9) 如果数据结构声明（包括数组、结构、类、枚举等）不是充分自注释的，必须加以注释。对数据结构的注释应放在其上方相邻位置，不可放在下面；对结构中的每个域的注释放在此域的右方。

(10) 全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及存取时注意事项等说明。

2. 注释

(11) 注释与所描述内容进行同样的缩排，可使程序排版整齐，并方便注释的阅读与理解。

(12) 将注释与其上面的代码用空行隔开。

(13) 对变量的定义和分支语句（条件分支、循环语句等）必须编写注释。因为这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释可以帮助更好地理解程序，有时甚至优于看设计文档。

2. 注释

(14) 对于switch语句下的case语句，如果因为特殊情况需要处理完一个case后进入下一个case处理，必须在该case 语句处理完、下一个case语句前加上明确的注释。这样比较清楚程序编写者的意图，有效防止无故遗漏break语句。

(15) 维护代码时，要更新相应的注释，删除不再有用的注释。保持代码、注释的一致性，避免产生误解。

3. 标识符命名

(1) 标识符的命名要清晰、明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。较短的单词可通过去掉“元音”形成缩写；较长的单词可取单词的头几个字母形成缩写；一些单词采用大家公认的缩写。

(2) 命名中若使用特殊约定或缩写，则要有注释说明。

(3) 自己特有的命名风格，要自始至终保持一致，不可来回变化。

3. 标识符命名

(4) 对于变量命名，建议除了要有具体含义外，还能表明其变量类型、数据类型等，因此最好不要用单个字符表示，如果用单个字符表示，很容易敲错，而编译时又检查不出来，有可能为了这个小小的错误而花费大量的查错时间。但使用单个字符表示局部循环变量是允许的（如i, j, k）。

(5) 命名规范必须与所使用的系统风格保持一致，并在同一项目中统一，比如采用UNIX的全小写加下划线的风格或大小写混排（如add_user或AddUser）的方式，不要使用大小写与下划线混排（如Add_User）的方式。

4. 可读性

(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。这是为了防止阅读程序时产生误解。

例如，本来是正确的代码：

```
If ( year % 4 == 0 || year % 100 != 0 && year % 400 == 0 )
```

如果加上括号，则更清晰。

```
If ((year % 4) == 0 || ((year % 100) != 0 && (year % 400) == 0))
```

4. 可读性

(2) 避免使用不易理解的数字，用有意义的标识来替代。

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。

5. 变量

(1) 去掉没必要的公共变量，以降低模块间的耦合度。

(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。

(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。这将有利于程序的进一步优化、单元测试、系统联调以及代码维护等。这种关系的说明可在注释或文档中描述。

(4) 当向公共变量传递数据时，要十分小心，若有必要应进行合法性检查，防止赋与不合理的值或越界等现象发生。

(5) 防止局部变量与公共变量同名。

(6) 严禁使用未经初始化的变量。特别是在C / C++中引用未经赋值的指针，经常会引起系统崩溃。

6. 函数

(1) 每个函数完成单一的功能，不设计多用途面面俱到的函数。

(2) 函数和过程中关系较为紧密的代码尽可能相邻。如初始化代码应放在一起，不应在中间插入实现其它功能的代码。

(3) 对所调用函数的错误返回码要仔细、全面地处理。

(4) 每个函数的源程序行数原则上应该少于200行。

对于消息分流处理函数，完成的功能统一，但由于消息的种类多，可能超过200行的限制，不属于违反规定。

6. 函数

(5) 编写可重入函数时，应注意局部变量的使用（如编写C / C++语言的可重入函数时，应使用auto即缺省态局部变量或寄存器变量），不应使用static局部变量，否则必须经过特殊处理，才能使函数具有可重入性。

(6) 编写可重入函数时，若使用全局变量，则应通过关中断、信号量（即P、V操作）等手段对其加以保护。若对所使用的全局变量不加以保护，则此函数就不具有可重入性，即当多个进程调用此函数时，很有可能使有关全局变量为不可知状态。

(7) 避免函数中不必要的语句，防止程序中的垃圾代码，预留代码应以注释的方式出现。

7. 可测试性

(1) 在同一项目组或产品组内，要有一套统一的为集成测试与系统联调准备的调测开关及相应打印函数，并且要有详细的说明。

(2) 在同一项目组或产品组内，调测打印出的信息串的格式要有统一的形式。信息串中至少要有所在模块名（或源文件名）和行号，以便于集成测试。

(3) 编程的同时要为单元测试选择恰当的测试点，并仔细构造测试代码、测试用例，同时给出明确的注释说明。测试代码部分应作为（模块中的）一个子模块，以方便测试代码在模块中的安装与拆卸（通过调测开关）

7. 可测试性

(4) 在进行集成测试 / 系统联调之前，要构造好测试环境、测试项目及测试用例，同时仔细分析并优化测试用例，以提高测试效率。好的测试用例应尽可能模拟出程序所遇到的边界值、各种复杂环境及一些极端情况等。

(5) 使用断言来发现软件问题，提高代码可测性。

(6) 使用断言来检查程序正常运行时不应发生，而在调测时有可能发生的非法情况。

(7) 不能用断言来检查最终产品肯定会出现且必须处理的错误情况。

7. 可测试性

(8)对较复杂的断言加上明确的注释。这样可澄清断言含义并减少不必要的误用。

(9)用断言确认函数的参数。

(10)用断言保证没有定义的特性或功能不被使用。

(11)用断言对程序开发环境（操作系统／编译器／硬件）的假设进行检查。

(12)正式软件产品中应把断言及其他调测代码去掉（即把有关调测开关关掉），可加快软件运行速度。

7. 可测试性

(13)用调测开关来切换软件的DEBUG版和正式版，而不要同时存在正式版本和DEBUG版本的不同源文件，以减少维护的难度。

(14)软件的DEBUG版本和发行版本应该统一维护，不允许分家，并且要时刻注意保证两个版本在实现功能上的一致性。

(15)在软件系统中设置与取消有关测试手段，不能对软件实现的功能等产生影响。即有测试代码的软件和关掉测试代码的软件，在功能行为上应一致。

8. 程序效率

(1) 在保证软件的正确性、稳定性、可读性及可测试性的前提下，提高代码效率。

(2) 局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响。

(3) 通过对系统数据结构的划分与改进，以及对程序算法的优化来提高空间效率。

(4) 循环体内工作量最小化。应仔细考虑循环体内的语句是否可以放在循环体之外，使循环体内工作量最小，从而提高程序的时间效率。

(5) 较大的局部变量(2K以上)应声明成静态类型(static)，避免占用太多的堆栈空间。避免发生堆栈溢出，出现不可预知的软件故障。

9. 质量保证

(1) 代码质量保证优先的原则：正确性，稳定性，安全性，可测试性，符合编码规范 / 可读性，系统整体效率，模块局部效率。

(2) 严禁使用未经初始化的变量。引用未经初始化的变量可能会产生不可预知的后果，特别是引用未经初始化的指针经常会导致系统崩溃，需特别注意。声明变量的同时初始化，除了能防止引用未经初始化的变量外，还可能生成更高效的机器代码。

(3) 定义公共指针的同时对其初始化。这样便于指针的合法性检查，防止应用未经初始化的指针。建议对局部指针也在定义的同时初始化，形成习惯。

(4) 只引用属于自己的存储空间。

9. 质量保证

(5) 防止引用已经释放的内存空间。在实际编程过程中，稍不留心就会出现在一个模块中释放了某个内存块(如指针)，而另一模块在随后的某个时刻又使用了它。要防止这种情况发生。

(6) 过程/函数中分配的内存，在过程/函数退出之前要释放。

(7) 过程 / 函数中申请的（为打开文件而使用的）文件句柄，在过程 / 函数退出要关闭。分配的内存不释放以及文件句柄不关闭，是较常见的错误，而且稍不注意就有可能发生。这类错误往往会引起很严重后果，且难以定位。

9. 质量保证

(8) 防止内存操作越界。所谓内存操作主要是指对数组、指计、内存地址等的操作。内存操作越界是软件系统主要错误之一，后果往往非常严重，所以当我们进行这些操作时一定要仔细小心。

(9) 系统运行之初要对加载到系统中的数据进行一致性检查。

(10) 严禁随便更改其他模块或系统的有关设置和配置。

(11) 不能随便改变与其他模块的接口。

(12) 充分了解系统的接口之后，再使用系统提供的功能。

9. 质量保证

(13) 编程时要防止关系运算符错误。如将“<=”误写成“<”或“>=”等造成的，由此引起的后果往往是很严重的，所以编程时，一定要在这些地方小心。当编完程序后，应对这些操作进行彻底检查。

(14) 要时刻注意混淆的操作符。当编完程序后，应从头至尾检查一遍这些操作符，以防止拼写错误。例如，如C++中的“=”与“==”、“|”与“||”、“&”与“&&”等，若拼写错了。编译器不一定能够检查出来。

9. 质量保证

(15) 有可能的话，if语句尽量加上else分支。switch语句必须有default分支。对不期望的情况(包括异常情况)进行处理，保证程序逻辑严谨。

(16) 减少没必要的指针使用，特别是较复杂的指针，如指针的指针、数组的指针，指针的数组，函数的指针等。

10. 代码编辑、编译、审查

(1) 打开编译器的所有告警开关对程序进行编译。

(2) 在产品软件（项目组）中。要统一编译开关选项。

(3) 通过代码走查及审查方式对代码进行检查。

(4) 测试部门产品之前，应对代码进行抽查及评审。

11. 代码测试、维护

(1) 单元测试要求至少达到语句覆盖。

(2) 单元测试开始要跟踪每一语句，并观察数据流及变量的变化。

(3) 清理、整理或优化后的代码要经过审查及测试。

(4) 代码版本升级要经过严格测试。

(5) 使用工具软件对代码版本进行维护。

(6) 正式版本上软件对代码版本都应有详细的文档记录。

12. 宏

(1) 用宏定义表达时，要使用完备的括号。
(2) 将宏定义的多条表达式放在在括号中。
(3) 使用宏时，不允许参数发生变化。

9.4 程序效率与性能分析

- **程序的效率是指程序的执行速度及程序所需占用内存的存储空间。**
- **程序编码是最后提高运行速度和节省存储的机会，因此在此阶段不能不考虑程序的效率。**

9.4 程序效率与性能分析

- 讨论程序效率的几条准则：

(1) 效率是一个性能要求，应当在需求分析阶段给出。软件效率以需求为准，不应以人力所及为准。

(2) 好的设计可以提高效率。

(3) 程序的效率与程序的简单性相关。

一般说来，任何对效率无重要改善，且对程序的简单性、可读性和正确性不利的程序设计方法都是不可取的。

算法对效率的影响

(1) 在编程序前，尽可能化简有关的算术表达式和逻辑表达式；

(2) 仔细检查算法中的嵌套的循环，尽可能将某些语句或表达式移到循环外面；

(3) 尽量避免使用多维数组；

(4) 尽量避免使用指针和复杂的表；

(5) 采用“快速”的算术运算；

(6) 不要混淆数据类型，避免在表达式中出现类型混杂；

(7) 尽量采用整数算术表达式和布尔表达式；

(8) 选用等效的高效率算法。

影响存储器效率的因素

- 这存储效率与操作系统的分页功能直接有关，并不是指要使所使用的存储空间达到最少。
- 采用结构化程序设计，将程序功能合理分块，使每个模块或一组密切相关模块的程序体积大小与每页的容量相匹配，可减少页面调度，减少内外存交换，提高存储效率。
- 提高存储器效率的关键是程序的简单性。

影响输入/输出的因素

(1) 输入 / 输出的请求应当最小化。
(2) 对于所有的输入 / 输出操作，安排适当的缓冲区，以减少频繁的信息交换。
(3) 对辅助存储（如磁盘），选择尽可能简单的、可接受的存取方法。
(4) 对辅助存储的输入，输出，应当成块传送。
(5) 对终端或打印机的输入 / 输出，应考虑设备特性，尽可能改善输入 / 输出的质量和速度。
(6) 任何不易理解的，对改善输入 / 输出效果关系不大的措施都是不可取的。
(7) 不应该为追求所谓“超高效”的输入 / 输出而损害程序的可理解性。
(8) 好的输入 / 输出程序设计风格对提高输入 / 输出效率会有明显的效果。

That's All!