1、选择合适的算法和数据结构选择一种合适的数据结构很重要，如果在一堆随机存放的数中使
用了大量的插入和删除指令，那使用链表要快得多。数组与指针语句具有十分密切的关系，一
般来说，指针比较灵活简洁，而数组则比较直观，容易理解。对于大部分的编译器，使用指针
比使用数组生成的代码更短，执行效率更高。在许多种情况下，可以用指针运算代替数组索引，
这样做常常能产生又快又短的代码。与数组索引相比，指针一般能使代码速度更快，占用空间
更少。使用多维数组时差异更明显。下面的代码作用是相同的，但是效率不一样数组索引
指针运算
For(;;){ p=arrayA=array[t++];
for(;;){ a=*(p++); 。。。。。。。。。。。。。。。。} }
指针方法的优点是，array 的地址每次装入地址 p 后，在每次循环中只需对 p 增量操作。在数
组索引方法中，每次循环中都必须根据 t 值求数组下标的复杂运算。

2、使用尽量小的数据类型能够使用字符型(char)定义的变量，就不要使用整型(int)变量来定
义；能够使用整型变量定义的变量就不要用长整型(longint)，能不使用浮点型(float)变量就
不要使用浮点型变量。当然，在定义变量后不要超过变量的作用范围，如果超过变量的范围赋
值,C 编译器并不报错,但程序运行结果却错了,而且这样的错误很难发现。在 ICCAVR 中，可
以在 Options 中设定使用 printf 参数，尽量使用基本型参数(%c、%d、%x、%X、%u 和%s 格式说
明符)，少用长整型参数(%ld、%lu、%lx 和%lX 格式说明符)，至于浮点型的参数(%f)则尽量不
要使用，其它 C 编译器也一样。在其它条件不变的情况下，使用%f 参数，会使生成的代码的数
量增加很多，执行速度降低。

3、减少运算的强度
（1）、查表(游戏程序员必修课)一个聪明的游戏大虾，基本上不会在自己的主循环里搞什么
运算工作，绝对是先计算好了，再到循环里查表。看下面的例子：
旧代码：
longfactorial(inti)

```
{
if(i==0)return1;
elsereturni*factorial(i-1);
}
```
新代码：
```
staticlongfactorial_table[]={1，1，2，6，24，120，720/*etc*/};
longfactorial(inti)
{
returnfactorial_table;
}
```
如果表很大，不好写，就写一个 init 函数，在循环外临时生成表格。
（2）、求余运算 a=a%8;可以改为：a=a&7;说明：位操作只需一个指令周期即可完成，而大
部分的 C 编译器的“%”运算均是调用子程序来完成，代码长、执行速度慢。通常，只要求是求
2n 方的余数，均可使用位操作的方法来代替。
（3）、平方运算 a=pow(a,2.0);可以改为：a=a*a;说明：在有内置硬件乘法器的单片机中(如
51 系列)，乘法运算比求平方运算快得多，因为浮点数的求平方是通过调用子程序来实现的，
在自带硬件乘法器的 AVR 单片机中，如 ATMega163 中，乘法运算只需 2 个时钟周期就可以完成。
既使是在没有内置硬件乘法器的 AVR 单片机中，乘法运算的子程序比平方运算的子程序代码短，
执行速度快。如果是求 3 次方，
如：a=pow(a，3。0)；
更改为：a=a*a*a；则效率的改善更明显。
（4）、用移位实现乘除法运算 a=a*4;b=b/4;可以改为：a=a<<2;b=b>>2;通常如果需要乘以或
除以 2n，都可以用移位的方法代替。在 ICCAVR 中，如果乘以 2n，都可以生成左移的代码，而
乘以其它的整数或除以任何数，均调用乘除法子程序。用移位的方法得到代码比调用乘除法子
程序生成的代码效率高。实际上，只要是乘以或除以一个整数，均可以用移位的方法得到结果，
如：a=a*9 可以改为：a=(a<<3)+a 采用运算量更小的表达式替换原来的表达式，下面是一个经
典例子:旧代码:
```
x=w%8;
y=pow(x，2.0)；
z=y*33;
```

```
for(i=0;i<MAX;i++)
{
h=14*i;printf("%d", h);
}
```
新代码:
```
x=w&7;/*位操作比求余运算快*/
y=x*x;/*乘法比平方运算快*/
z=(y<<5)+y;/*位移乘法比乘法快*/
for(i=h=0;i<MAX;i++)
{
h+=14;/*加法比乘法快*/
printf("%d", h);
}
```
（5）、避免不必要的整数除法整数除法是整数运算中最慢的，所以应该尽可能避免。一种可
能减少整数除法的地方是连除，这里除法可以由乘法代替。这个替换的副作用是有可能在算乘
积时会溢出，所以只能在一定范围的除法中使用。
不好的代码：
```
inti, j, k, m;
m=i/j/k;
```
推荐的代码：
```
inti, j, k, m;
m=i/(j*k);
```
（6）、使用增量和减量操作符在使用到加一和减一操作时尽量使用增量和减量操作符，因为
增量符语句比赋值语句更快，原因在于对大多数 CPU 来说，对内存字的增、减量操作不必明显
地使用取内存和写内存的指令，比如下面这条语句：x=x+1;模仿大多数微机汇编语言为例，产
生的代码类似于：moveA, x;把 x 从内存取出存入累加器 AaddA, 1;累加器 A加
lstorex;
把新值存回 x 如果使用增量操作符，生成的代码如下：incrx;x 加 1 显然，不用取指令和存
指令，增、减量操作执行的速度加快，同时长度也缩短了。
（7）、使用复合赋值表达式复合赋值表达式(如 a-=1 及 a+=1 等)都能够生成高质量的程序代
码
（8）、提取公共的子表达式
在某些情况下，C++编译器不能从浮点表达式中提出公共的子表达式，因为这意味着相当于对表达式重新排序。需要特别指出的是，编译器在提取公共子表达式前不

能按照代数的等价关系重新安排表达式。这时，程序员要手动地提出公共的子表达式（在 VC.NET 里有一项“全局优化”选项可以完成此工作，但效果就不得而知了）。
不好的代码：

```
float a， b， c， d， e， f;
。。。
e = b * c / d;
f = b / d * a;
```

推荐的代码：

```
float a， b， c， d， e， f;
。。。
const float t(b / d);
e = c * t;
f = a * t;
```

不好的代码：

```
float a， b， c， e， f;
。。。
e = a / c;
f = b / c;
```

推荐的代码：

```
float a， b， c， e， f;
。。。
const float t(1.0f / c);
e = a * t;
f = b * t;
```

4、结构体成员的布局
很多编译器有“使结构体字，双字或四字对齐”的选项。但是，还是需要改善结构体成员的对齐，有些编译器可能分配给结构体成员空间的顺序与他们声明的不同。但是，有些编译器并不提供这些功能，或者效果不好。所以，要在付出最少代价的情况下实现最好的结构体和结构体成员对齐，建议采取下列方法：
（1）按数据类型的长度排序
把结构体的成员按照它们的类型长度排序，声明成员时把长的类型放在短的前面。编译器要求把长型数据类型存放在偶数地址边界。在申明一个复杂的数据类型（既有多字节数据又有单字节数据）时，应该首先存放多字节数据，然后再存放单字节数据，这样可以避免内存的空洞。编译器自动地把结构的实例对齐在内存的偶数边界。
（2）把结构体填充成最长类型长度的整倍数
把结构体填充成最长类型长度的整倍数。照这样，如果结构体的第一个成员对齐了，所有整个结构体自然也就对齐了。下面的例子演示了如何对结构体成员进行重新排序：
不好的代码，普通顺序：

```
struct
{
    char a[5];
    long k;
    double x;
} baz;
```
推荐的代码，新的顺序并手动填充了几个字节：
```
struct
{
    double x;
    long k;
    char a[5];
    char pad[7];
} baz;
```
这个规则同样适用于类的成员的布局。

（3）按数据类型的长度排序本地变量

当编译器分配给本地变量空间时，它们的顺序和它们在源代码中声明的顺序一样，和上一条规则一样，应该把长的变量放在短的变量前面。如果第一个变量对齐了，其它变量就会连续的存放，而且不用填充字节自然就会对齐。有些编译器在分配变量时不会自动改变变量顺序，有些编译器不能产生 4 字节对齐的栈，所以 4 字节可能不对齐。下面这个例子演示了本地变量声明的重新排序：

不好的代码，普通顺序
```
short ga， gu， gi;
long foo， bar;
double x， y， z[3];
char a， b;
float baz;
```
推荐的代码，改进的顺序
```
double z[3];
double x， y;
long foo， bar;
float baz;
short ga， gu， gi;
```
（4）把频繁使用的指针型参数拷贝到本地变量

避免在函数中频繁使用指针型参数指向的值。因为编译器不知道指针之间是否存在冲突，所以指针型参数往往不能被编译器优化。这样数据不能被存放在寄存器中，而且明显地占用了内存带宽。注意，很多编译器有“假设不冲突”优化开关（在 VC 里必须手动添加编译器命令行/Oa 或/Ow），这允许编译器假设两个不同的指针总是有不同的内容，这样就不用把指针型参数保存到本地变量。否则，请在函数一开始把指针指向的数据保存到本地变量。如果需要的话，在函数结束前拷贝回去。

不好的代码：

```
// 假设 q != r
void isqrt(unsigned long a,  unsigned long* q,  unsigned long* r)
{
    *q = a;
    if (a > 0)
    {
        while (*q > (*r = a / *q))
        {
            *q = (*q + *r) >> 1;
        }
    }
    *r = a - *q * *q;
}
```

推荐的代码：

```
// 假设 q != r
void isqrt(unsigned long a,  unsigned long* q,  unsigned long* r)
{
    unsigned long qq,  rr;
    qq = a;
    if (a > 0)
    {
        while (qq > (rr = a / qq))
        {
            qq = (qq + rr) >> 1;
        }
    }
    rr = a - qq * qq;
    *q = qq;
    *r = rr;
}
```

5、循环优化

（1）、充分分解小的循环

要充分利用 CPU 的指令缓存，就要充分分解小的循环。特别是当循环体本身很小的时候，分解循环可以提高性能。注意:很多编译器并不能自动分解循环。 不好的代码：

```
// 3D 转化：把矢量 V 和 4x4 矩阵 M 相乘
for (i = 0;  i < 4;  i ++)
{
    r[i] = 0;
    for (j = 0;  j < 4;  j ++)
```

```
    {
        r[i] += M[j][i]*V[j];
    }
}
```

推荐的代码：

```
r[0] = M[0][0]*V[0] + M[1][0]*V[1] + M[2][0]*V[2] + M[3][0]*V[3];
r[1] = M[0][1]*V[0] + M[1][1]*V[1] + M[2][1]*V[2] + M[3][1]*V[3];
r[2] = M[0][2]*V[0] + M[1][2]*V[1] + M[2][2]*V[2] + M[3][2]*V[3];
r[3] = M[0][3]*V[0] + M[1][3]*V[1] + M[2][3]*V[2] + M[3][3]*v[3];
```

（2）、提取公共部分

对于一些不需要循环变量参加运算的任务可以把它们放到循环外面，这里的任务包括表达式、函数的调用、指针运算、数组访问等，应该将没有必要执行多次的操作全部集合在一起，放到一个 init 的初始化程序中进行。

（3）、延时函数

通常使用的延时函数均采用自加的形式：

```
void delay (void)
{
    unsigned int i;
    for (i=0;i<1000;i++) ;
}
```

将其改为自减延时函数：

```
void delay (void)
{
    unsigned int i;
    for (i=1000;i>0;i--) ;
}
```

两个函数的延时效果相似，但几乎所有的 C 编译对后一种函数生成的代码均比前一种代码少 1~3 个字节，因为几乎所有的 MCU 均有为 0 转移的指令，采用后一种方式能够生成这类指令。在使用 while 循环时也一样，使用自减指令控制循环会比使用自加指令控制循环生成的代码更少 1~3 个字母。但是在循环中有通过循环变量“i”读写数组的指令时，使用预减循环有可能使数组超界，要引起注意。

（4）、while 循环和 do…while 循环

用 while 循环时有以下两种循环形式：

```
unsigned int i;
i=0;
while (i<1000)
{
    i++;
    //用户程序
}
```

或：

```
unsigned int i;
i=1000;
do
{
    i--;
    //用户程序
}
while (i>0);
```

在这两种循环中，使用 do…while 循环编译后生成的代码的长度短于 while 循环。

（6）、循环展开

这是经典的速度优化，但许多编译程序(如 gcc -funroll-loops)能自动完成这个事，所以现在你自己来优化这个显得效果不明显。

旧代码:

```
for (i = 0; i < 100; i++)
{
    do_stuff(i);
}
```

新代码:

```
for (i = 0; i < 100; )
{
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
}
```

可以看出，新代码里比较指令由 100 次降低为 10 次，循环时间节约了 90%。不过注意:对于中间变量或结果被更改的循环，编译程序往往拒绝展开，(怕担责任呗)，这时候就需要你自己来做展开工作了。

还有一点请注意，在有内部指令 cache 的 CPU 上(如 MMX 芯片)，因为循环展开的代码很大，往往 cache 溢出，这时展开的代码会频繁地在 CPU 的 cache 和内存之间调来调去，又因为 cache 速度很高，所以此时循环展开反而会变慢。还有就是循环展开会影响矢量运算优化。

（6）、循环嵌套

把相关循环放到一个循环里，也会加快速度。

旧代码:

```
for (i = 0; i < MAX; i++) /* initialize 2d array to 0's */
for (j = 0; j < MAX; j++)
a[i][j] = 0.0;
for (i = 0; i < MAX; i++) /* put 1's along the diagonal */
a[i][i] = 1.0;
```

新代码:

```
for (i = 0; i < MAX; i++) /* initialize 2d array to 0's */
{
    for (j = 0; j < MAX; j++)
    a[i][j] = 0.0;
    a[i][i] = 1.0; /* put 1's along the diagonal */
}
```

（7）、Switch 语句中根据发生频率来进行 case 排序

Switch 可能转化成多种不同算法的代码。其中最常见的是跳转表和比较链/树。当 switch 用比较链的方式转化时，编译器会产生 if-else-if 的嵌套代码，并按照顺序进行比较，匹配时就跳转到满足条件的语句执行。所以可以对 case 的值依照发生的可能性进行排序，把最有可能的放在第一位，这样可以提高性能。此外，在 case 中推荐使用小的连续的整数，因为在这种情况下，所有的编译器都可以把 switch 转化成跳转表。

不好的代码:

```
int days_in_month,  short_months,  normal_months,  long_months;
。。。。。。
switch (days_in_month)
{
    case 28:
    case 29:
        short_months ++;
        break;
    case 30:
        normal_months ++;
        break;
    case 31:
        long_months ++;
        break;
    default:
        cout << "month has fewer than 28 or more than 31 days" << endl;
        break;
}
```

推荐的代码:

```
int days_in_month,  short_months,  normal_months,  long_months;
。。。。。。
```

```
switch (days_in_month)
{
    case 31:
        long_months ++;
        break;
    case 30:
        normal_months ++;
        break;
    case 28:
    case 29:
        short_months ++;
        break;
    default:
        cout << "month has fewer than 28 or more than 31 days" << endl;
        break;
}
```

（8）、将大的 switch 语句转为嵌套 switch 语句

当 switch 语句中的 case 标号很多时，为了减少比较的次数，明智的做法是把大 switch 语句转为嵌套 switch 语句。把发生频率高的 case 标号放在一个 switch 语句中，并且是嵌套 switch 语句的最外层，发生相对频率相对低的 case 标号放在另一个 switch 语句中。比如,下面的程序段把相对发生频率低的情况放在缺省的 case 标号内。

```
pMsg=ReceiveMessage();
switch (pMsg->type)
{
    case FREQUENT_MSG1:
    handleFrequentMsg();
    break;
    case FREQUENT_MSG2:
    handleFrequentMsg2();
    break;
    。。。。。。
    case FREQUENT_MSGn:
    handleFrequentMsgn();
    break;
    default: //嵌套部分用来处理不经常发生的消息
    switch (pMsg->type)
    {
        case INFREQUENT_MSG1:
        handleInfrequentMsg1();
        break;
```

```
        case INFREQUENT_MSG2:
        handleInfrequentMsg2();
        break;
        。。。。。。
        case INFREQUENT_MSGm:
        handleInfrequentMsgm();
        break;
    }
}
```

如果 switch 中每一种情况下都有很多的工作要做，那么把整个 switch 语句用一个指向函数指针的表来替换会更加有效，比如下面的 switch 语句，有三种情况：

```
enum MsgType{Msg1， Msg2， Msg3}
switch (ReceiveMessage()
{
    case Msg1;
    。。。。。。
    case Msg2;
    。。。。。
    case Msg3;
    。。。。。
}
```

为了提高执行速度，用下面这段代码来替换这个上面的 switch 语句。

```
/*准备工作*/
int handleMsg1(void);
int handleMsg2(void);
int handleMsg3(void);
/*创建一个函数指针数组*/
int (*MsgFunction [])()={handleMsg1， handleMsg2， handleMsg3};
/*用下面这行更有效的代码来替换 switch 语句*/
status=MsgFunction[ReceiveMessage()]();
```

（9）、循环转置

有些机器对 JNZ(为 0 转移)有特别的指令处理，速度非常快，如果你的循环对方向不敏感，可以由大向小循环。

旧代码:

```
for (i = 1; i <= MAX; i++)
{
    。。。
}
```

新代码:

```
i = MAX+1;
while (--i)
```

```
{
    。。。
}
```
不过千万注意，如果指针操作使用了 i 值，这种方法可能引起指针越界的严重错误(i = MAX+1;)。当然你可以通过对 i 做加减运算来纠正，但是这样就起不到加速的作用，除非类似于以下情况：
旧代码:
```
char a[MAX+5];
for (i = 1; i <= MAX; i++)
{
    *(a+i+4)=0;
}
```
新代码:
```
i = MAX+1;
while (--i)
{
    *(a+i+4)=0;
}
```
（10）、公用代码块
一些公用处理模块，为了满足各种不同的调用需要，往往在内部采用了大量的 if-then-else 结构，这样很不好，判断语句如果太复杂，会消耗大量的时间的,应该尽量减少公用代码块的使用。(任何情况下，空间优化和时间优化都是对立的--东楼)。当然，如果仅仅是一个(3==x)之类的简单判断，适当使用一下，也还是允许的。记住，优化永远是追求一种平衡，而不是走极端。
（11）提升循环的性能
要提升循环的性能，减少多余的常量计算非常有用（比如，不随循环变化的计算）。
不好的代码(在 for()中包含不变的 if())：
```
for( i 。。。 )
{
    if( CONSTANT0 )
    {
        DoWork0( i );  // 假设这里不改变 CONSTANT0 的值
    }
    else
    {
        DoWork1( i );  // 假设这里不改变 CONSTANT0 的值
    }
}
```
推荐的代码：
```
if( CONSTANT0 )
{
```

```
    for( i 。。。 )
    {
        DoWork0( i );
    }
}
else
{
    for( i 。。。 )
    {
        DoWork1( i );
    }
}
```

如果已经知道 if()的值，这样可以避免重复计算。虽然不好的代码中的分支可以简单地预测，但是由于推荐的代码在进入循环前分支已经确定，就可以减少对分支预测的依赖。

（12）、选择好的无限循环

在编程中，我们常常需要用到无限循环，常用的两种方法是 while (1) 和 for (;;)。这两种方法效果完全一样，但那一种更好呢？然我们看看它们编译后的代码：

编译前：

```
while (1);
```

编译后：

```
mov eax, 1
test eax, eax
je foo+23h
jmp foo+18h
```

编译前：

```
for (; ; );
```

编译后：

```
jmp foo+23h
```

显然，for (; ;)指令少，不占用寄存器，而且没有判断、跳转，比 while (1)好。

6、提高 CPU 的并行性

（1）使用并行代码

尽可能把长的有依赖的代码链分解成几个可以在流水线执行单元中并行执行的没有依赖的代码链。很多高级语言，包括 C++，并不对产生的浮点表达式重新排序，因为那是一个相当复杂的过程。需要注意的是，重排序的代码和原来的代码在代码上一致并不等价于计算结果一致，因为浮点操作缺乏精确度。在一些情况下，这些优化可能导致意料之外的结果。幸运的是，在大部分情况下，最后结果可能只有最不重要的位（即最低位）是错误的。

不好的代码：

```
double a[100],  sum;
```

```
int i;
sum = 0.0f;
for (i=0;  i<100;  i++)
sum += a[i];
```
推荐的代码：
```
double a[100],  sum1,  sum2,  sum3,  sum4,  sum;
int i;
sum1 = sum2 = sum3 = sum4 = 0.0;
for (i = 0;  i < 100;  i += 4)
{
    sum1 += a[i];
    sum2 += a[i+1];
    sum3 += a[i+2];
    sum4 += a[i+3];
}
sum = (sum4+sum3)+(sum1+sum2);
```
要注意的是：使用 4 路分解是因为这样使用了 4 段流水线浮点加法，浮点加法的每一个段占用一个时钟周期，保证了最大的资源利用率。

（2）避免没有必要的读写依赖

当数据保存到内存时存在读写依赖，即数据必须在正确写入后才能再次读取。虽然 AMD Athlon 等 CPU 有加速读写依赖延迟的硬件，允许在要保存的数据被写入内存前读取出来，但是，如果避免了读写依赖并把数据保存在内部寄存器中，速度会更快。在一段很长的又互相依赖的代码链中，避免读写依赖显得尤其重要。如果读写依赖发生在操作数组时，许多编译器不能自动优化代码以避免读写依赖。所以推荐程序员手动去消除读写依赖，举例来说，引进一个可以保存在寄存器中的临时变量。这样可以有很大的性能提升。下面一段代码是一个例子：

不好的代码：
```
float x[VECLEN],  y[VECLEN],  z[VECLEN];
。。。。。。
for (unsigned int k = 1;  k < VECLEN;  k ++)
{
    x[k] = x[k-1] + y[k];
}
for (k = 1;  k <VECLEN;  k++)
{
    x[k] = z[k] * (y[k] - x[k-1]);
}
```
推荐的代码：
```
float x[VECLEN],  y[VECLEN],  z[VECLEN];
。。。。。。
float t(x[0]);
```

```
for (unsigned int k = 1;  k < VECLEN;  k ++)
{
    t = t + y[k];
    x[k] = t;
}
t = x[0];
for (k = 1;  k <;  VECLEN;  k ++)
{
    t = z[k] * (y[k] - t);
    x[k] = t;
}
```

7、循环不变计算

对于一些不需要循环变量参加运算的计算任务可以把它们放到循环外面，现在许多编译器还是能自己干这件事，不过对于中间使用了变量的算式它们就不敢动了，所以很多情况下你还得自己干。对于那些在循环中调用的函数，凡是没必要执行多次的操作通通提出来，放到一个 init 函数里，循环前调用。另外尽量减少喂食次数，没必要的话尽量不给它传参，需要循环变量的话让它自己建立一个静态循环变量自己累加，速度会快一点。

还有就是结构体访问，东楼的经验，凡是在循环里对一个结构体的两个以上的元素执行了访问，就有必要建立中间变量了(结构这样，那 C++的对象呢?想想看)，看下面的例子:

旧代码:

```
total =
a->b->c[4]->aardvark +
a->b->c[4]->baboon +
a->b->c[4]->cheetah +
a->b->c[4]->dog;
```

新代码:

```
struct animals * temp = a->b->c[4];
total =
temp->aardvark +
temp->baboon +
temp->cheetah +
temp->dog;
```

一些老的 C 语言编译器不做聚合优化，而符合 ANSI 规范的新的编译器可以自动完成这个优化，看例子:

```
float a,  b,  c,  d,  f,  g;
。。。
a = b / c * d;
f = b * g / c;
```

这种写法当然要得，但是没有优化

```
float a， b， c， d， f， g;
。。。
a = b / c * d;
f = b / c * g;
```

如果这么写的话，一个符合 ANSI 规范的新的编译器可以只计算 b/c 一次，然后将结果代入第二个式子，节约了一次除法运算。

8、函数优化

（1）Inline 函数

在 C++中，关键字 Inline 可以被加入到任何函数的声明中。这个关键字请求编译器用函数内部的代码替换所有对于指出的函数的调用。这样做在两个方面快于函数调用：第一，省去了调用指令需要的执行时间；第二，省去了传递变元和传递过程需要的时间。但是使用这种方法在优化程序速度的同时，程序长度变大了，因此需要更多的 ROM。使用这种优化在 Inline 函数频繁调用并且只包含几行代码的时候是最有效的。

（2）不定义不使用的返回值

函数定义并不知道函数返回值是否被使用，假如返回值从来不会被用到，应该使用 void 来明确声明函数不返回任何值。

（3）减少函数调用参数

使用全局变量比函数传递参数更加有效率。这样做去除了函数调用参数入栈和函数完成后参数出栈所需要的时间。然而决定使用全局变量会影响程序的模块化和重入，故要慎重使用。

（4）所有函数都应该有原型定义

一般来说，所有函数都应该有原型定义。原型定义可以传达给编译器更多的可能用于优化的信息。

（5）尽可能使用常量(const)

尽可能使用常量(const)。C++ 标准规定，如果一个 const 声明的对象的地址不被获取，允许编译器不对它分配储存空间。这样可以使代码更有效率，而且可以生成更好的代码。

（6）把本地函数声明为静态的(static)

如果一个函数只在实现它的文件中被使用，把它声明为静态的(static)以强制使用内部连接。否则，默认的情况下会把函数定义为外部连接。这样可能会影响某些编译器的优化——比如，自动内联。

9、采用递归

与 LISP 之类的语言不同，C 语言一开始就病态地喜欢用重复代码循环，许多 C 程序员都是除非算法要求，坚决不用递归。事实上，C 编译器们对优化递归调用一点都不反感，相反，它们还很喜欢干这件事。只有在递归函数需要传递大量参数，可能造成瓶颈的时候，才应该使用循环代码，其他时候，还是用递归好些。

10、变量

（1）register 变量

在声明局部变量的时候可以使用 register 关键字。这就使得编译器把变量放入一个多用途的寄存器中，而不是在堆栈中，合理使用这种方法可以提高执行速度。函数调用越是频繁，越是可能提高代码的速度。

在最内层循环避免使用全局变量和静态变量，除非你能确定它在循环周期中不会动态变化，大多数编译器优化变量都只有一个办法，就是将他们置成寄存器变量，而对于动态变量，它们干脆放弃对整个表达式的优化。尽量避免把一个变量地址传递给另一个函数，虽然这个还很常用。C 语言的编译器们总是先假定每一个函数的变量都是内部变量，这是由它的机制决定的，在这种情况下，它们的优化完成得最好。但是，一旦一个变量有可能被别的函数改变，这帮兄弟就再也不敢把变量放到寄存器里了，严重影响速度。看例子：

```
a = b();
c(&d);
```

因为 d 的地址被 c 函数使用，有可能被改变，编译器不敢把它长时间的放在寄存器里，一旦运行到 c(&d)，编译器就把它放回内存，如果在循环里，会造成 N 次频繁的在内存和寄存器之间读写 d 的动作，众所周知，CPU 在系统总线上的读写速度慢得很。比如你的赛杨 300，CPU 主频 300，总线速度最多 66M，为了一个总线读，CPU 可能要等 4-5 个周期，得。。得。。得。。想起来都打颤。

（2）、同时声明多个变量优于单独声明变量

（3）、短变量名优于长变量名，应尽量使变量名短一点

（4）、在循环开始前声明变量

11、使用嵌套的 if 结构

在 if 结构中如果要判断的并列条件较多，最好将它们拆分成多个 if 结构，然后嵌套在一起，这样可以避免无谓的判断。