# C语言常见优化策略

## # 基本优化

### 整体思路与误区

当前编译器的优化其实已经做了很多工作，很多时候我们想当然的任务更优的代码，实际上在编译器的优化下，它的汇编指令基本一致的。编译器优化功能对那些平铺直叙的代码更有效，避免在编码里面加入一些想当然的”花招“，这反而会影响编译器优化。
（性能优化优先级：系统设计>数据结构/算法选择>热点代码编码调整）

1. 全局变量：全局变量绝不会位于寄存器中。使用指针或者函数调用，可以直接修改全局变量的值。因此，编译器不能将全局变量的值缓存在寄存器中，但这在使用全局变量时便需要额外的（常常是不必要的）读取和存储。所以，在重要的循环中我们不建议使用全局变量。如果函数过多的使用全局变量，比较好的做法是拷贝全局变量的值到局部变量，这样它才可以存放在寄存器。这种方法仅仅适用于全局变量不会被我们调用的任意函数使用。
2. 用switch()函数替代if…else…
3. 使用二分方式中断代码而不是让代码堆成一列
4. 带参数的宏定义效率比函数高。简单的运算可以用宏定义来完成。
5. 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。

6. 能使用指针操作的尽量使用指针操作，一般来说，指针比较灵活简洁，对于大部分的编译器，
   使用指针生成的代码更短，执行效率更高。
7. 递归调用尽量换成内循环或者查表解决，因为频繁的函数调用也是很浪费资源的查表是数据结构中的一个概念。查表的前提是先建表。在C语言实现中，建表也就是将一系列的数据，或者有原始数据中提取出的特征值，存储到一定的数据结构中，如数组或链表中。查表的时候，就是对数组或链表查询的过程。常用的方式有如下几种：
   - 对于有序数组，可以采用折半查找的方式快速查询。
   - 对于链表，可以根据链表的构建方式，进行针对性查询算法的编写。
   - 大多数情况，可以通过遍历的方式进行查表。即从第一个元素开始，一直顺序查询到最后一个元素，逐一对比。
8. 使用增量或减量操作符：`++x`；原因是增量符语句比赋值语句更快。
9. 使用复合赋值表达式：`x+=1`；能够生成高质量的程序代码
10. 代码中使用代码块可以及时回收不再使用的变量，提高性能。变量的作用域从定义变量的那一行代码开始，一直到所在代码块结束。
11. 当一个函数被调用很多次，而且函数中某个变量值是不变的，应该将此变量声明为static（只会分配一次内存），可以提高程序效率。
12. 循环：长循环在内，短循环在外。

# 移位实现乘除

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：

```
1 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);
}
```

避免不必要的整数除法也是优化的策略。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:
int i,  j,  k,  m;
m = i / j / k;
//推荐的代码:
int i,  j,  k,  m;
m = i / (j * k);
```

# # 结构体成员的布局

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;
```

```
17 //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;
}
```

# 循环优化

循环优化整体策略为循环展开、循环合并与循环顺序的交换：

**循环展开，降低循环层次或者次数**

```
while(i < count){
  a[i]=i;
  i++;
}
//优化为:
while(i < count - 1){
  a[i]=i;
  a[i+1] = i+1;
  i += 2;
}
if(i==count-1){
  a[count-1]=count-1;
}
```

## 循环合并（计数器相同的），避免多次轮询

```
if(i = 0; i < index; i++){
  do_type_a_work(i);
}
if(i = 0; i < index; i++){
  do_type_b_work(i);
}

//优化为:
if(i = 0; i < index; i++){
  do_type_a_work(i);
  do_type_b_work(i);
}
```

## 循环顺序交换

循环内计算外提（每次计算不变），降低无效计算:

```
for(int i=0; i< get_max_index();i++){}
//优化为:
int max_index = get_max_index();
for(int i = 0; i < max_index; i++){}
```

循环内多级寻址外提，避免反复寻址跳转:

```
for(int i = 0; i < max_index; i++){
  ainfo->bconfig.cset[i].index = index;
  ainfo->bconfig.cset[i].flag = flag;
}

//优化:
set = ainfo->bconfig.cset;
for(int i = 0; i < max_index; i++){
  set[i].index = index;
  set[i].flag = flag;
}
```

循环内判断外提（某时刻结果不变），降低无效比较次数:

```
if (i = 0; i < index; i++) {
  if (type==TYPE_A) {
    do_type_a_work(i);
  } else {
    do_type_b_work(i);
    }
}
// 优化:
if (type==TYPE_A) { // 提高性能的同时，影响了可维护性;
  if (i = 0; i < index; i++) {
    do_type_a_work(i);
  }
} else {
  if (i = 0; i < index; i++) {
    do_type_b_work(i);
```

```
  }
}
```

循环体使用int类型，多重循环：最忙的循环放最里面

```
for (column = 0; column < 100; column ++) {
  for (row = 0; row < 5; row++) {
    sum += table[row][column ];
  }
}
// 优化
for (row = 0; row < 5; row++) {
  for (column = 0; column < 100; column ++) {
    sum += table[row][column ];
  }
}
```

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。

5. 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。
6. 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。
7. 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。

8. 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)。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。

9. 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。

10. 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。

# # 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;
}
```

# # 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;
优化后代码:
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。

# 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。避免小函数调用开销（提炼宏函数 或 inline内联化）

```
int afunc(char *buf, bool enable){
  if(check_null(buff)==true){
    return -1;
  }
}

//优化为:
#define check_null(a) if (a==null){return -1;}
int afunc(char *buf, bool enable){
  check_null(buf);
}
```

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。函数入参低于一定数量（如4个），则形参由R0,R1,R2,R3四个寄存器进行传递；若形参个数大于的部分必须通过堆栈进行传递，性能降低；用指针传递的效率高于结构赋值；**直接用全局变量省去了传递时间，但是影响了模块化和可重入，要慎重使用；**
   如果函数不需要返回值就明确void；
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。

```
int max(int *a, int m, int n);//这行就是函数原型，函数定义在主函数后面。
//函数原型的就是实现函数先（main中调用），
//后（定义在后面）
```

5. 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。

# # 变量

减少不必要的赋值或者变量初始化，减少不必要的临时变量；

```
int i = 0;
i = input_value;
//优化:
int i = input_value
```

```
int ret = do_next_level_func();
return ret;
//优化:
return do_next_level_func();
```

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。
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。
5. 如果确定整数非负，应直接使用 unsigned int ，处理器处理无符号unsigned 整形数的效率远远高于有符号signed整形数。
6. 局部变量尽可能的 不使用char和short类型 。对于char和short类型，编译器需要在每次赋值的时候将局部变量减少到8或者16位，是通过寄存器左移24或者16位，然后根据有无符号标志右移相同的位数实现，这会消耗两次计算机指令操作。

7. 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 （long int），能 不使用浮点型 （float）变量就不要使用浮点型变量。

# 条件判断

1. 使用`switch`替代`if else`：switch…case会生成一份大小（表项数）为最大case常量+1的跳表，程序首先判断switch变量是否大于最大case 常量，若大于，则跳到default分支处理；否则取得索引号为switch变量大小的跳表项的地址（即跳表的起始地址+表项大小*索引号），程序接着跳到此地址执行。
2. 在`if(xxx1>XXX1 && xxx2=XXX2)`多个条件判断中，确保AND表达式的第一部分最快或最早得到结果，这样第二部分便有可能不需要执行。
3. 在必须使用if…else…语句，将最可能执行的放在最前面。
4. 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。

> 整理自：
>
> https://blog.csdn.net/runafterhit/article/details/107677483
>
> https://www.cnblogs.com/ggzhangxiaochao/p/13962852.html
>
> https://jingyan.baidu.com/article/8ebacdf0730c0f49f65cd500.html
>
> https://blog.csdn.net/sunjiajiang/article/details/7887724?utm_medium=distribute.pc_relevant.none-task-blog-title-2&spm=1001.2101.3001.4242
>
> https://segmentfault.com/a/1190000037447486