C&C++实效编程百例

求是科技　编著

人民邮电出版社

图书在版编目(CIP)数据

C&C++实效编程百例 / 求是科技编. —北京：人民邮电出版社，2004.3

ISBN 7-115-12102-8

I．C… II．求… III．C 语言—程序设计 IV．TP312

中国版本图书馆 CIP 数据核字(2004)第 011886 号

内容提要

本书精选了 124 个最具代表性的 C 和 C++语言学习和开发的编程实例，包括了基础应用、字符串处理、数组、指针与引用、类与对象、函数、重载、数据结构与算法、模板、组件与泛型设计、图形界面外观、磁盘文件、系统与硬件、网络与通信、数据库、设计模式等内容。

本书所选实例在突出其实用性的同时，也侧重帮助读者理解 C 和 C++的重点以及难懂的概念。

本书适合正在学习 C 和 C++语言进行实际开发的人员阅读，帮助读者很好地理解重点概念，迅速掌握实际应用中的各种经验、技巧。

C&C++实效编程百例

◆ 编　　著　求是科技
责任编辑　张立科

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
读者热线　010-671940942
北京汉魂图文设计有限公司制作
北京顺义向阳胶印厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092　1/16
印张：23.25
字数：715 千字　　2004 年 3 月第 1 版
印数：1 –　000 册　　2004 年 3 月北京第 1 次印刷

ISBN 7-115-12102-8/TP・3863

定价：35.00 元（附光盘）

本书如有印装质量问题，请与本社联系　电话：（010）67129223

前　言

C 和 C++是编程语言中的经典，一个好的程序开发人员应具备良好的 C 和 C++编程基础乃至开发技巧。本书通过 124 个最具代表性的编程实例详细讲解 C 和 C++重点知识和开发技巧。希望帮助读者很好地理解重点概念，迅速掌握实际应用中的各种经验、技巧。

本书实例的分类按照完成功能来划分，尽量将相关的内容归纳在一章中。通过每章的实例，向读者讲解了有关基础应用、字符串处理、数组、指针与引用、类与对象、函数、重载、数据结构与算法、模板、组件与泛型设计、图形界面外观、磁盘文件、系统与硬件、网络与通信、数据库、设计模式等内容。本书是广大 C 和 C++学习和使用人员积累编程经验、拓展视野的得力助手。本书的一部分实例强调了实用性，这些实例占本书绝大部分比例，另一部分实例侧重帮助读者理解重点、难懂概念，用最简单的代码说明最关键的问题。

本书大部分实例的讲解都分为 3 个步骤：

- 实例目的——讲解本例的功能所在，指出本例要到达的目的和效果，让读者做到心中有数。
- 实现方法——讲解技术原理或设计思路，给出技术原理的合理解释、规范的算法和流程描述，便于读者阅读代码、学习程序设计方法。
- 程序代码——给出具体的实现过程，包括界面设计、编写代码和注释，读者可参照实现。

在完成本书时，得到了很多老师和网友的大力支持，提供给本书诸多好的、经典源码，在此表示诚挚的谢意。同时，本书极少部分内容参阅的一些网络资源未能找到原作者，在此也表示对原著作者最衷心的感谢。

由于时间、水平限制，书中缺点和不足之处在所难免，敬请读者批评指正。

编者

2004.02

目 录

第 1 章　基础应用

- C++层次代码优化
- C++的数据抽象
- 定义 C++的标志位
- 源代码的命名规范和书写规范
- 用类型定义精简代码的后期调整

实例 1　C++层次代码优化

实例目标

谈到代码优化，很多人都会直接想到汇编语言。难道优化仅限于汇编语言吗？当然不是，利用 C++一样可以进行代码优化操作，而且有些往往能收到意想不到的效果。下面通过具体的实例，让读者了解在 C++中是如何进行代码优化的。

实现方法和程序代码

代码的优化可以从以下几个方面考虑。

1．确定浮点型变量及表达式是 float 型

为了让编译器生成更好的代码（比如说生成 3DNow!或 SSE 指令的代码），必须确定浮点型变量和表达式统一为 float 型。这里要特别注意的是，浮点型常量以“F”或“f”为后缀才是 float 型（如 3.14f），否则默认为 double 型。在函数声明时，应使用 float 关键字来声明函数及其参数，防止浮点型参数被默认为是 double 类型。

2．使用 32 位的数据类型

C++的编译器虽然有很多种，但它们都包含的典型的 32 位数据类型有：int、signed、signed int、unsigned、unsigned int、long、signed long、long int、signed long int、unsigned long，unsigned long int。在编写 C++程序时，应尽量采用这些统一的 32 位数据类型。

3．适当选取带符号或无符号的整型变量

在很多情况下，需要考虑整型变量是带符号还是无符号类型。比如，一个人的体重数据不可能出现负数，所以相应变量不需要使用带符号类型，但如果要保存温度数据，就必须使用带符号的变量。

在编写程序时，需要考虑是否选用带符号的变量。在某些情况下，带符号类型的数据运算比较快，但有时却相反。

例如：从大于 16 位的整型数据向浮点型转化时，使用带符号整型比较快。因为 x86 构架中提供了从带符号整型转化为浮点类型的指令，而没有提供从无符号整型转化到浮点类型的指令。作为两者的对比，下面举例列出一组“不好的代码”和“推荐代码”，它们编译后的结果分别如表 1-1 和表 1-2 所示。

表 1-1　　不好的代码

编译前	编译后
double x;	mov [temp + 4], 0
unsigned int i;	mov eax, i
x = i;	mov [temp], eax
	flid qword ptr [temp]
	fstp qword ptr [x]

上面的代码比较慢。不但因为指令数目比较多，而且指令匹配失败造成的 FLID 指令被延迟执行。如表 1-2 所示的是优化的代码。

表 1-2　　推荐的代码

编译前	编译后
double x;	fild dword ptr [i]

续表

编译前	编译后
int i;	fstp qword ptr [x]
x = i;	

在整数运算中计算商和余数时，使用无符号类型比较快。表 1-3 比较了带符号和无符号的整型数除以 4 时编译器分别产生的代码。

表 1-3　　代码的比较

不好的代码		推荐的代码	
编译前	编译后	编译前	编译后
int i;	mov eax, i	unsigned int i;	shr i, 2
i = i / 4;	cdq	i = i / 4;	
	and edx, 3		
	add eax, edx		
	sar eax, 2		
	mov i, eax		

一般来说，无符号类型应用于以下的一些情况：

- 除法和余数
- 循环计数
- 数组下标

带符号类型应用的情况为：整型到浮点的转化。

4．while 与 for 的比较

在编程中，常常需要用到无限循环结构，常用的两种方法是 while(1)和 for(;;)。这两种方法效果完全一样，编译之后的代码如表 1-4 所示。

表 1-4　　**while** 与 **for** 比较

编译前	编译后
While (1);	mov eax,1
	test eax,eax
	je temp+23h
	jmp temp+18h
for (;;);	jmp temp+23h

可以看出，for(;;)在编译后指令少，不占用寄存器而且无需判断跳转，比 while (1)好。

5．适当使用数组型结构代替指针型数据

指针型的数据对编译器来说难于优化，因为目前尚无有效的指针代码优化方法，编译器总是假设指针变量可以访问内存的任意地方，包括分配给其他变量的储存空间。所以为了使编译器更好地优化代码，要避免在不必要的地方使用指针类型。一个很常见的例子是：在访问存放在数组中的数据时，C++允许使用操作符[]或指针两种方式来访问数组元素。使用数组型代码会让编译器减少产生不安全代码的可能性。比如，x[0]和 x[2]不可能是同一个内存地址，但*p 和*q 可能。这里推荐使用数组类型，因为这样可能会有一定程度的性能的提升。下面以矩阵和向量的乘法为例进行说明，如表 1-5 所示。

表 1-5　　数组型结构代替指针型数据的代码示例

不好的代码（针型数据）	推荐的代码（数组型结构）
typedef struct {	typedef　struct {

续表

不好的代码（指针型数据）

```
float x,y,z,w;
} VERTEX;
typedef struct
{
    float m[4][4];
} MATRIX;
void XForm(float* res, const float* v, const float* m,
int nNumVerts)
{
    float dp;
    int i;
    const VERTEX* vv = (VERTEX *)v;
    for (i = 0; i < nNumVerts; i++)
    {
        dp = vv->x * *m ++;
        dp += vv->y * *m ++;
        dp += vv->z * *m ++;
        dp += vv->w * *m ++;
        *res ++ = dp;            // 写入转换了的 x
        dp = vv->x * *m ++;
        dp += vv->y * *m ++;
        dp += vv->z * *m ++;
        dp += vv->w * *m ++;
        *res ++ = dp;     // 写入转换了的 y
        dp = vv->x * *m ++;
        dp += vv->y * *m ++;
        dp += vv->z * *m ++;
        dp += vv->w * *m ++;
        *res ++ = dp;     // 写入转换了的 z
        dp = vv->x * *m ++;
        dp += vv->y * *m ++;
        dp += vv->z * *m ++;
        dp += vv->w * *m ++;
        *res ++ = dp;            // 写入转换了的 w
        vv ++;                   // 下一个矢量
        m -= 16;
    }
}
```

推荐的代码（数组型结构）

```
    float x,y,z,w;
} VERTEX;
typedef  struct
{
    float m[4][4];
} MATRIX;
void XForm (float* res, const float* v, const float* m,
int nNumVerts)
{
    int i;
    const VERTEX* vv = (VERTEX*)v;
    const MATRIX* mm = (MATRIX*)m;
    VERTEX* rr = (VERTEX*)res;
    for (i = 0; i < nNumVerts; i++)
    {
        rr->x = vv->x * mm->m[0][0] + vv->y *
mm->m[0][1]
                + vv->z * mm->m[0][2] + vv->w *
mm->m[0][3];
        rr->y = vv->x * mm->m[1][0] + vv->y *
mm->m[1][1]
                + vv->z * mm->m[1][2] + vv->w *
mm->m[1][3];
        rr->z = vv->x * mm->m[2][0] + vv->y *
mm->m[2][1]
                + vv->z * mm->m[2][2] + vv->w *
mm->m[2][3];
        rr->w = vv->x * mm->m[3][0] + vv->y *
mm->m[3][1]
                + vv->z * mm->m[3][2] + vv->w *
mm->m[3][3];
        vv++;
        rr++;
    }
}
```

注意：源代码向机器码的转化是与编译器的代码发生器相关的。从源代码层次很难控制产生的机器码。在利用一些编译器编译特殊的源代码时，有可能指针型代码编译后的机器码比同等条件下的数组型代码运行速度更快。因此最明智的做法是比较源代码在转化之后的性能哪个更高，再选择使用指针型还是数组型。

6．小循环的完全展开

要充分利用 CPU 的指令缓存，就要把小的循环完全展开。当循环体很小的时候，展开循环可以提高性能。此外，很多编译器不具备自动展开循环的功能。如表 1-6 所示，给出了小循环和循环展开的示例代码。

表 1-6　小循环和循环展开的示例代码

不好的代码（小循环）	推荐的代码（循环展开）

// 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];
}
}
	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];

7．避免没有必要的读写依赖

当数据保存到内存时会存在读写依赖。AMD Athlon 等 CPU 有减少读写依赖延迟的硬件，允许在数据写入内存前把它读取出来。但是，如果能够避免读写依赖并把数据保存在内部寄存器中，读写速度会更快。在一段很长而又互相依赖的代码链中，避免读写依赖显得尤其重要。在读写依赖发生在数组操作的情况下，许多编译器不能自动优化代码去避免读写依赖，所以推荐读者尝试手动消除读写依赖。举例来说，引进一个可以保存在寄存器中的临时变量，就能获得很大的性能提升。下面举一个例子，如表 1-7 所示。

表 1-7　临时变量的有误，导致性能差异的示例代码

不好的代码（无临时变量）	推荐的代码（引入临时变量）

double x[VECLEN], y[VECLEN], z[VECLEN];
unsigned int k;
for (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]);
}
	double x[VECLEN], y[VECLEN], z[VECLEN];
	unsigned int k;
	double t = x[0];
	for (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;
	}

8．switch 的用法

在 switch 结构中，推荐按照 case 的值发生的可能性进行排序，把最有可能的放在第一个，当 switch 结构用比较链的方式转化时，可以提高性能。另外，在 case 中推荐使用小的连续整数，因为在这种情况下，所有的编译器都可以把 switch 结构转化成跳转表。例如 12 个月中大月的可能性最大，故可将 31 排在首位，其代码比较如表 1-8 所示。

表 1-8　Switch 语句优劣比较

不好的代码	推荐的代码
int days_in_month, short_months, normal_months, long_months; switch (days_in_month) {	int days_in_month, short_months, normal_months, long_months; switch (days_in_month) {

续表

不好的代码	推荐的代码

case 28:
case 29:
short_months ++;
break;
case 30:
normal_months ++;
break;
case 31:
long_months ++;
break;
default:
printf ("month has fewer than 28 or more than 31 days\n");
break;
}
	case 31:
	long_months ++;
	break;
	case 30:
	normal_months ++;
	break;
	case 28:
	case 29:
	short_months ++;
	break;
	default:
	printf ("month has fewer than 28 or more than 31 days\n");
	break;
	}

9．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。

10．尽可能使用常量（const）

尽可能使用常量（const）的定义方式。这种方式可提高代码效率，而且可以生成更好的代码，因为附加的信息对编译有利。比如，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。

11．把本地函数声明为静态的（Static）

如果一个函数在实现它的文件外未被使用的话，则应把它声明为静态的（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。

12．考虑动态内存分配

动态内存分配（C++中的 new 语句）可能总是为大的基本类型（四字对齐）返回一个对齐的指针。但是如果不能保证对齐，则使用以下代码来进行“四字对齐”操作。四字对齐是指四字（如 double 型变量）的数据存储按 8 字节对齐，避免由于硬件引起的在内存中的重复存储。这段代码假设指针可以映射到 long 型。

```
double* p = (double*)new BYTE[sizeof(double) * number_of_doubles+7L];
double* np = (double*)((long(p) + 7L) & –8L);
```

现在，可以使用 np 代替 p 来访问数据。

> 注意：p 在释放储存空间时仍然需要。

13．使用显式的并行代码

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。这一点对浮点代码尤其重要，不管它被映射成 x87 或 3DNow!指令，因为浮点操作有很长的潜伏期。很多高级语言，包括 C++，并不对产生的浮点表达式重新排序。如表 1-9 所示，给出了将长的有依赖的代码进行分解的示例。

表 1-9　将长的有依赖的代码进行分解的示例

不好的代码	推荐的代码
double a[100], sum;	double a[100], sum1, sum2, sum3, sum4, sum;

int I;
sum = 0.0f;
for (I=0; I<100; I++)
　　sum += a[I];
	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);

注意：重排序代码和原来的代码在代数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。在大部分情况下，最后结果可能只有最不重要的位是错误的。

注意：使用 4 路分解是因为使用了 4 阶段流水线浮点加法，浮点加法的每一个阶段占用一个时钟周期，保证了最大的资源利用率。

14．提出公共子表达式

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，因为这相当于对表达式重新排序。尤其是，编译器在提取共用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，程序员要手动提出公共的子表达式。注意，重新安排表达式可能出现不同的计算结果，但这些结果通常只是最不重要的位有误差。如表 1-10 所示，为使用公共子表达式的示例。

表 1-10　应用公共子表达式的示例

不好的代码	推荐的代码

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

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

15．结构体成员的布局

很多编译器有使结构体字，双字或 4 字对齐的选项。另外，还需要改善结构体成员的对齐情况，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，有些编译器可能并不提供这些功能，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，建议采取这些方法。

16．按类型长度排序

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。

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。Double 类型的长度为 8、long 类型的长度为 4、char 类型的长度为 1，原结构体总长度 17，将其长度填充为 24 并将 double 类型放在第一位。如表 1-11 所示，为变量声明时排序不同的比较示例。

表 1-11　　为变量声明时排序不同的比较示例

不好的代码（随意排序）	推荐的代码（充分考虑排序）
struct { 　char a[5]; 　long k; 　double x; } baz;	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。

17．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，其他变量就会连续地存放，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 4Byte 对齐的栈，所以 4Byte 可能不对齐。表 1-12 所示的例子演示了本地变量声明的重新排序。

表 1-12　　演示了本地变量声明的重新排序

不好的代码，普通顺序	推荐的代码，改进的顺序
short ga, gu, gi; long foo, bar; double x, y, z[3]; char a, b; float baz;	double z[3]; double x, y; long foo, bar; float baz; short ga, gu, gi;

18．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。一种可以减少整数除法的方案是避免连除，以乘法代替。如表 1-13 所示，给出了避免不必要的整数除法代码示例。

表 1-13　　避免不必要的整数除法代码示例

不好的代码	推荐的代码
int I, j, k, m; m = I / j / k;	int I, j, k, m; m = I / (j * k);

注意：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。

19．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。这样数据就不能被存放在寄存器中，而且明显地占用了内存空间。注意，很多编译器有“假设不冲突”优化开关，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，应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，可以在函数结束前恢复该指针原来指向的数据值。如表 1-14 所示，为将频繁使用的指针型参数

拷贝到本地变量的代码示例。

表 1-14　将频繁使用的指针型参数拷贝到本地变量的代码示例

不好的代码	推荐的代码

// 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;
}
	// 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;
	}

20．赋值与初始化

先看看以下代码。

```
class CInt
{
        int m_i;

public:
        CInt(int a = 0):m_i(a) { cout << "CInt" << endl; }
        ~CInt() { cout << "~CInt" << endl; }

        CInt operator + (const CInt& a) { return CInt(m_i + a.GetInt()); }

        void SetInt(const int i)        { m_i = i; }
        int GetInt() const      { return m_i; }
};
```

这里定义了一个 CInt 类，表 1-15 中给出两组代码，分别调用这个 CInt 类。

表 1-15　调用同一类实现相同功能，但调用方式不同的代码示例

不好的代码	推荐的代码

void main()
{
CInt a, b, c;
a.SetInt(1);
b.SetInt(2);
c = a + b;
}
	void main()
	{
	CInt a(1), b(2);
	CInt c(a + b);
	}

两段代码实现的功能是一样的，从输出结果中能发现，不好的代码输出了 4 个“CInt”和 4 个“~CInt”，而推荐的代码只输出 3 个。也就是说，第二个例子比第一个例子少生成一次临时对象。

注意：第一个例子中的变量 c 用的是先声明再赋值的方法，第二个用的是用初始化的方法，它