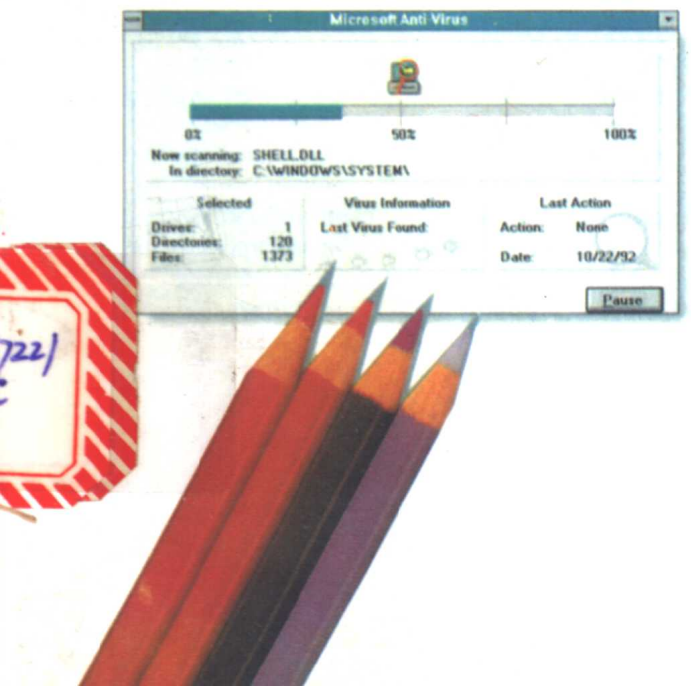
Windows技术丛书之六

MS-DOS 和 Windows 环境下的 C / C++编程技术

郝金川　潘　青
陈　洁　景　利　编

海洋出版社

6



Windows 软件技术丛书之六

MS-DOS 和 Windows 环境下的 C/C++编程技术

海 洋 出 版 社

1996 年 · 北京

内 容 简 介

本书对C/C++的程序设计技术进行了全面而详细地讲解。本书共十二章和一个附录，共分为两大部分：第一部分讲解如何编写高效率的C/C++程序，并提供有关优化的信息，且描述了在开发过程中能缩短编译时间的预编译头文件。第二部分描述了图形功能和QuickWin库，并说明了如何用混合语言进行程序设计。以及实现可移植程序的方法和设计技术。本书叙述通俗易懂，强调C/C++程序设计的特点，注重方法和技巧的描述，有助于培养程序员良好的的程序设计风格。

本书既适用于从事计算机软件开发的技术人员，亦可供在校师生及自学者使用。

Windows 软件技术丛书之六

MS-DOS 和 Windows 环境下的 C/C++ 编程技术

郝金川　潘　青
陈　浩　景　利　编
陈一飞　刘　忠　审校

*　　*　　*

海洋出版社出版（北京复兴门外大街1号）
海洋出版社发行　　兰空印刷厂印刷
开本：787×1092毫米 1/16　印张：13.6　字数：310千字
1993年12月第一版　　1996年5月第二次印刷
印数：1—5000册　　定价：193.00元/套（6册）
ISBN7－5027－3821－5/TP・234

目　录

第二部分 专用环境

简　介

《C／C++编程技术》一书叙述如何利用 C／C++特性编写程序．本书讨论的主题包括语言扩充、专用库函数以及程序设计策略与编译程序选项之间的相互影响．

本书不是一本有关 C／C++实用工具的参考书．所以，如果用户有关于 Code View 调试程序、程序员工作台（PWB）或者任何命令行实用程序的专门问题，请参阅有关参考书籍或联机帮助．

《C／C++编程技术》一书分为两个部分．第一部分"改进程序性能"帮助用户编写高效率的程序．这部分提供了有关优化的专用信息，即何时以及为什么使用各种不同的优化选项并且描述了在开发过程中能缩短编译时间的预编译头文件．第一部分解释了如何将用户程序编译成 P 代码（P-code），一种可以生成可执行文件的代码形式,并且说明了 C 和 C++所使用的存储管理选项以及使用的时机．第六章描述内嵌式汇编程序，这种特性使用户可在 C 和 C++源程序代码中使用汇编语言．第七章描述浮点数学包．

第二部分"专用环境"叙述了图形功能和 Quick Win 库,并且说明了如何用混合语言进行程序设计，以及实现可移植程序的方法．C 运行库包含用于低级图形操作的图形函数，例如画直线、矩形和圆．库中还包含用于建立直观图形的函数，例如扇形（饼形）图表和横方图．C／C++还包括一个库，可以使用户把带有简单输入和输出要求的 DOS 程序转换成窗口程序．

注：术语"DOS"指 MS-DOS 和 IBM PC DOS 操作系统．当需要提示某种系统特有的特性时，使用明确的操作系统名字．

本书使用如下约定：

举例	说明
STDIO.H	大写字母表示文件名、段名、寄存器和用于操作系统命令级的术语．
[[option]]	双方括号中的项是可选项．
#pragma pack {1|2}	大括号和竖线表示从两个或更多项中选择一项．除非大括号外还括有双方括号，否则就必须选择一项．
#include <io.h>	这种字体用于文本中举例、用户输入、程序输出和错误消息．
CL [[option…]] file…	某项后面的三个点（省略号）表示可后接多个相同格式的项．

while()
{
.
.
.
}
	垂直的三个点表示程序举例中有意省略的部分．

CTR+ENTER	用来表示键盘上的键名．当在两个键名当中有加号（+）时，应按

	住第一个键，并按第二个键．回车键（有时在键盘上用弯箭头表示）叫作 ENTER．
"argument"	在书中第一次定义的术语用引号括起．
"C string"	有些C结构（例如字符串）需要使用引号．

第一部分　改进程序性能

C／C++使用其完善的优化程序和增强的存储管理能力帮助用户建立最快、最小的应用程序.

第一章介绍各种优化使用的时机，并叙述了C／C++如何产生执行速度最快、占用空间最小的高效程序代码．第二章描述能显著缩短编译时间的特性——预编译头文件．第三章解释如何将用户程序编译成P代码，这是一类生成较小可执行文件的代码．第四章和第五章介绍C／C++提供给用户在C和C++中分配和管理程序存储空间的工具，包括___based类型和虚拟存储管理程序．如果用户程序需要做局部优化，可以使用第六章中介绍的内嵌式汇编程序，以产生尽可能紧凑的代码．第七章解释如何选择C／C++数学包，以帮助有浮点运算应用需要的用户．

第一章　程序优化

C／C++编译程序把 C 或 C++源语句翻译为机器可执行的指令．此外，编译程序还重写或优化程序的某些部分，使程序执行起来效率更高．这种优化在源程序一级不很明显．

编译程序通常进行三种类型的优化：1) 修改或移动代码区，目的是减少所使用的指令数，或使所用的指令更加有效地利用处理器；2) 移动代码并合成操作，最大限度地利用寄存器，因为在处理机寄存器中进行数据操作比在内存中进行同样操作要快得多；3) 删除冗余或未用的代码区．

本章介绍用 C／C++编译程序控制代码优化的各种方法．

1.1　从程序员工作台 (PWB)控制优化

程序员工作台 (Programmer's WorkBench，PWB)是一个集成开发环境，用于编辑、建立和调试用 C 或 C++编写的应用程序．

在 PWB 中进行编译的方法有两种：第一种,Debug 编译．在缺省的 Debug 编译方式下，编译程序不执行任何优化；第二种,Release 编译．在缺省的 Release 编译方式下，编译程序执行大部分优化．

要选择编译程序在 Debug 或 Release 编译方式下所执行的优化，可调出 Options (选择)菜单，从中选出 Language Options (语言选择)子菜单．从该菜单中，打开 C 或 C++ Compiler Options (编译程序选择) 对话框．用该对话框，可以指定 debug 或 release 编译，然后可以打开优化对话框，选择独特的优化．

在每一个 Compiler Options 对话框中的优化都对应于 CL 的一个命令行选项．(事实上，PWB 即根据输入内容建立一个命令行并把它传递给 CL.)

注：在本章中，我们根据优化的效果、调用优化的命令行选项以及控制优化的编译指示 (pragma) 来讨论优化选择．所有这些优化可以使用 Compiler Options 对话框在编译单位 (文件) 级加以控制．

1.2　从命令行控制优化

从命令行控制优化要求明确应用程序需要何种优化．然后，使用／O(有些情况下为／G) 开始的命令行选项指定这些优化．

如果多个选择项之间发生冲突，编译程序就采用命令行中指定的最后一个选项．命令行

```
CL      /Oa      /OL      /Ot      TEST.C
```

表示编译 TEST.C 程序．它说明编译程序可以做到：

1) 在假定没有别名的情况下进行优化（/Oa）.

2) 执行循环优化（/Ol）.

3) 执行其它一般性的加速优化（/Ot）.

上述命令行亦可写成：

```
CL      /Oalt      TEST.C
```

1.3 用编译指示控制优化

有时，需要对编译程序优化进行精细控制．命令行选项允许就整个编译单位（文件）控制优化．此外，C/C++支持多种编译指示，可以以单个函数为基础进行优化控制．

在本章中，控制优化的编译指示按优化效果分类加以描述．

使用编译指示 optimize，可以按单个函数为基础控制下列各个优化参数：

1) 与别名代码有关的行为（a 和 w）.

2) 内嵌式函数调用（b0，b1，或 b2）.

3) 局部公共子表达式的缩减（c）.

4) 全局公共子表达式的缩减（g）.

5) 全局寄存器分配（e）.

6) 循环优化（l）.

7) 最佳优化（x）.

8) 强度优化（z）.

9) 禁止不安全的优化（n）.

10) 保持浮点结果的一致性（p）.

11) 为每一个函数安排一个出口（r）.

12) 精简代码或加快执行速度的优化（s 或 t）.

还有一个把程序编译为 P 代码（P-code）的选项（/Oq），以及一些只有当 P 代码启用时才能应用的选项（/Of，/Of-，/Ov，及/Ov-）．关于这些选项的情况参见第三章"用 P 代码精简程序"．

任何优化或选项的合成均可用编译指示 optimize 启动或取消．例如，如果某函数大量使用了别名，现在不想优化由于别名可能产生问题的代码部分，同时又想优化那些未使用别名的代码段，可使用如下编译指示 optimize：

```
/* Function(s) that do not do aliasing. */
.
.
.
#pragma optimize( "a", off )
/* Function(s) that do aliasing. */
.
.
```

```
#pragma optimize( "a", on )
/ * More function(s) that do not do aliasing. * /
```

编译指示 optimize 的一些参数可以合成到一个串中，以便于一次打开或关闭多个选项。如:

```
#pragma optimize ("lge", off)
```

就禁止循环优化、全局公共子表达式优化和全局寄存器分配。

1.4 缺省优化

除了／Od（禁止优化）和／f（快速编译）选项外其它命令行选项都不能显式地禁止某些缺省优化。这些优化涉及范围较小，通常很有用。它们包括：删除小范围的公共子表达式，死存储的删除以及常量传递

1.4.1 公共子表达式删除

删除公共子表达式时，编译程序首先找到含有重复子表达式的代码，然后修改代码，令子表达式只计算一次。删除子表达式通常使用临时变量，如下例所示:

```
a = b+c * d;
x = c * d / y;
```

这两行中含有公共子表达式 c * d。该代码可在经过修改后只计算一次 d * d; 结果放入一个临时变量中（通常为一寄存器):

```
tmp = d * d;
a = b+tmp;
x = tmp / y;
```

1.4.2 死存储删除

死存储删除是公共子表达式删除的延伸。在一小段代码中含有同样值的变量可以合并到一个临时变量中。

在下列代码段中，编译程序发现表达式 func (x) 等值于 func (a+b):

```
x = a+b;
x = func (x) ;
```

这样，编译程序可以重写该代码如下:

```
x = func (a+b) ;
```

1.4.3 常量传递

在进行常量传递时，编译程序先分析变量赋值语句，然后确定是否它们可以变为常量

赋值语句．在下面的例子中，当将 i 赋给 j 时，i 为 7:

```
i=7;
j=i;
```

现在不把 i 赋给 j，而直接把常量 7 赋给 j:

```
i=7;
j=7;
```

虽然在源文件中也可以做这些改变，但却可能降低程序的可读性．而在多数情况下，优化不仅仅是提高程序效率，而且可在不降低效率的情况下写出更易读的代码．

在使用符号调试程序前不进行优化．在有些情况下，可能需要禁止优化，甚至包括缺省优化．因为优化会重新安排目标文件中的代码，从而在调试时使某些代码的识别发生困难．故通常在使用符号调试程序之前，最好不进行任何优化．可以用／Od（禁止优化）选择项全面禁止优化，或使用／f（快速编译）选择．

可以使用语句#pragma optimize（""，off）禁止对某一函数的任何优化．若要恢复优化到以前的状态，可使用语句#pragma optimize（""，on）．

1.5 特定优化项的选择

对许多应用程序而言，缺省优化就足够了，但有时用户要求编译程序进行一些特定优化．优化选项提供了一种方法，可以在优化代码时给编译程序提供具体目标．

1.5.1 选择速度或长度优化（／Ot 和／Os）

除了缺省优化外，C／C++编译程序还自动使用提高速度的优化选项／Ot．／Ot 选项允许的优化既能提高速度，亦能增加程序量．如果要为缩短程序长度而优化，那么使用／Os 选项．／Os 选项缩短了程序长度，但同时可能降低程序运行速度．

若以函数为单位进行速度或长度优化，可使用带 t 选项的优化编译指示．on 指示编译程序进行速度优化; off 指示编译程序优化应令代码更紧凑．例如:

```
#pragma optimize( "t", off )          /* Optimize for smallest code. */
.
.
.
#pragma optimize( "t", on )           /* Optimize for fastest code. */
```

使用／Oq 选项（P 代码生成）时隐含有／Os 选项．

1.5.2 生成内部函数（／Oi）

在某些正常函数调用的地方，C／C++编译程序可能插入运行速度更快的"内部函数"(intrinsic function)．每一次调用函数时，就执行一组指令存储参数，并为局部变量生成空间．函数返回时，必须执行更多的代码，以释放局部变量和参数所占用的空间，并把值

返回给调用函数的例行程序．这些指令执行起来需要时间．就中等大小的函数而言，这些附加代码（即指令）相对较少，但如果函数本身只有一两行，则附加的代码可能会占去函数编译后代码的一半．

要想避免这种代码扩展就要避免这类短函数，特别是在速度极为关键的常用代码区中．然而，许多库函数恰恰只有一两行代码．为此，编译程序提供了两种特定的库函数形式．一种形式是标准 C 函数，它需要与函数调用同样多的开销．另一种形式是一组指令，执行与函数同样的操作却不需要发出调用函数．第二种形式即所谓内部函数．内部函数比调用的函数速度快，在目标代码级能提供极好的优化效果．

例如，函数 strcpy 可以编写如下：

```
int strcpy  (char * dest, char * source)
{
    while  ( * dest ++ = * source++) ;
}
```

编译程序含有 strcpy 的内部形式．如果用户指示编译程序生成内部函数，那么对 strcpy 的任何调用都将被该内部形式取代．

注：为清楚起见，上面的例子是用 C 编写的，但大多数库函数为充分利用 80X86 指令集都是用汇编语言编写的．而内部函数并非简单地定义为宏的库函数．

选择 / Oi 进行编译可使编译程序使用下列函数的内部形式：

abs	labs	memset	strcat
_disable	_lrotl	outp	strcmp
_enable	_lrotr	_outpw	strcpy
_inp	memcmp	_rotl	strlen
_inpw	memcpy	_rotr	_strset

虽然下列浮点函数没有真正的内部形式，它们却有一种形式可以直接地把自变量传递到浮点运算器，而不是将参数压进通常的参数栈内：

acos	fmod	_acosl	_fmodl
asin	log	_asinl	_logl
atan	log10	_atanl	_log10l
atan2	pow	_atan2l	_powl
ceil	sin	_ceill	_sinl
cos	sinh	_cosl	_sinhl
cosh	sqrt	_coshl	_sqrtl
exp	tan	_expl	_tanl
floor	tanh	_floorl	_tanhl

注意： 编译程序执行优化的前提是内部数学函数没有副作用．如果用户已编写了自己的__matherr 函数，且该函数将改变全局变量，那么，这种前提就不能成立．因此，如果用户编写了自己的__matherr 函数来处理浮点错误，且此函数会产生副作用，那么应使用编译指示 function，使编译程序不生成数学函数

的内部代码.

如果想让编译程序只为以上所列函数的一个子集生成内部函数，那么可以使用 intrinsic 编译指示而不要用 / Oi 选择项. intrinsic 编译指示格式如下:

#pragma intrinsic (function1, …)

如果想对以上函数的大部分生成内部函数，只对小部分进行函数调用，那么可以采用 / Oi 选项进行编译，用 function 编译指示强制函数的使用. function 编译指示格式如下:

#pragma function (function1, …)

下列代码说明了 intrinsic 编译指示的用法:

```
#pragma intrinsic(abs)

void main( void )
{
    int i, j;

    i = big_routine_1();
    j = abs( i );
    big_routine_2( j );
}
```

为该程序生成的内部函数会导致取数绝对值的汇编语言代码代替对 abs 的调用. 由于调用 abs 时没有函数调用开销，程序将执行得更快.

在上个例子中，速度增加总体上说还比较小，因为只有一次对 abs 的调用. 在下面的例子中，abs 调用在循环中进行，有多次调用，因而通过生成内部函数可以节省大量执行时间.

```
#pragma intrinsic( abs )
void main( void )
{
int i, j, x;

    for( j=0; j<1000; j++ )
    {
    for( i=0; i<1000; i++ )
    {
        x += abs( i-j );
    }
    }
    printf( "The value of x is %d\n", x );
}
```

下列各点是使用函数调用内部形式的一些限制:

1) 浮点数字函数的内部形式不能与替换数学库（m LIBCAy. LIB）一起使用.

2) 如果使用了／Oq 选项（P-code 生成），则无法使用／Oi 选项.

3) 如果使用了／Ox选项（最大优化），等于启动了／Oi（生成内部函数）选项. 故在使用／Ox 时要当心，不要与上述各点发生冲突.

1.5.3 内嵌式函数调用（inlining function call）（／Ob0，／Ob1，／Ob2）

内嵌式用法类似于内部函数，只是前者不局限于库函数的具体集合. 内嵌式用法允许编译程序在函数被调用的每一个地方嵌入它的一个拷贝. 这样就消除了函数调用的开销，但函数的多个拷贝又会使程序变大.

可以通过关键字___inline（在 C++中为 inline）加以说明，把一个函数显式地记为直接嵌入的候选者. 任何在类（class）说明中定义的 C++成员（member）函数都被隐式地当作内嵌式函数.

是否执行直接嵌入要根据编译程序的判断. 若一个长度仅仅几行的函数被说明为内嵌式函数，那么编译程序可以忽视___inline 关键字. ／Obn 选项控制编译程序执行直接嵌入的程度.

／Ob0 选项禁止所有的直接嵌入，即便显式地说明为内嵌式函数的函数亦不例外. 当指定／Od 时,它为缺省.

／Ob1 选项根据编译程序的判断扩展所有说明为内嵌式的函数. 当／Od 未被指定时它为缺省.

根据编译程序的判断，／Ob2 选项扩展所有说明为内嵌式函数以及编译程序认为适合直接嵌入的任何其它函数.

1.5.4 假设无别名（／Oa 和／Ow）

／Oa 和／Ow 选项控制编译程序执行优化时根据"别名使用"情况所做的假设. 这些选项可以显著地改善程序性能，但有些场合却不很适用.

当一个存储单元使用了不只一个名字的时候，就是发生"别名使用"情况. 例如:

```
char      c;
char      * cptr;

cptr = &c;                    / * Take the address of c * /
                              / * c now has two names: c and * cptr * /
c = 1;                        / * Use first name * /
* cptr = 2;                   / * Use second name * /
```

表达式 * cptr 就是 C 的一个别名，因为它是这个变量的又一个名字. 当同时用两个名字指向一个变量时，就是在使用别名.

编译程序使用的优化技术之一就是把常用的变量存储在寄存器中，因为访问寄存器的速度比存取存储单元要快. 如果编译程序发现了别名，它就不再把变量放入寄存器，因为通过别名进行修改可能导致该变量所用的值不一致. 以上述代码段为例，若编译程序把 C 放入寄存器，且又修改了 * cptr，则 C 的值将会不一致: 一个存储在寄存器中，另一个存

在存储单元中。为避免出现这个问题，编译程序不把 C 放入寄存器中。

如上所述编译程序可以发现简单的别名使用情况。然而，编译程序并不能识别别名的所有可能形式。为安全起见，编译程序一般假设用户程序可能使用了它无法识别的别名。这意味着编译程序假设用户每次通过指针修改一个存储单元时，也可能修改了下列的值项：1) 任何全局变量;2) 地址已被取走的任何局部变量;3) 被另一个指针指向的存储单元。

这种假设限制了编译程序可能执行的优化程度。

/Oa 选择项通知编译程序用户程序不存在使用别名的情况（编译程序可以识别的简单形式不算在内）。这就使编译程序可以更充分地对代码进行优化。然而，如果指定此选项去编译确有别名的程序，则编译程序可能产生错误代码。下例中的程序就使用了别名而编译程序未能发现。当选用 /Oa 编译时，该程序便产生了错误结果：

```
/* OATEST.C
 * Fails when compiled with \0a.
 * Passes when compiled with default optimization.
 */
#include <stdio.h>
char buf[10];                          /* Global arrray */

char * return__buf()
{
    return buf;
}

void main()
{
    char * first,
     * second;
    first = buf;
    second = return__buf();
     * first = 2;
     * second = 3;
    if( * first == 3 )
    printf("pass\n");
    else
    printf("Fail\n");
}
```

在本例中，* first 和 * second 都指向同一存储单元。通过它的每一个名字，该单元被赋与两个不同值。如果该程序不用 /Oa 选项进行编译，则编译程序假设 * second 与 * first 可能指向同一存储单元，且 * first 可能被语句 * second = 3 修改。然而，如果用户确实指定了 /Oa 选项，编译程序则假设 * first 和 * second 指向不同的存储单元。编译程序接着假设 * first 值为 2，因此它跳过 if 语句而直接转向 else 子句，于是打印出“Fail”(失败)。