第一部分

语　法　篇

本部分内容

第 1 章　从 C 继承而来的

第 2 章　从 C 到 C++，需要做出一些改变

第 3 章　说一说“内存管理”的那点事儿

第 4 章　重中之重的类

第 5 章　用好模板，向着 GP 开进

第 6 章　让神秘的异常处理不再神秘

第 7 章　用好 STL 这个大轮子

第 1 章　从 C 继承而来的

C 和 C++ 可以说是所有编程语言中关系最为紧密的两个。在目标上，C++ 被定位为“a better C”；在名称上，C++ 有一个乳名叫做“C with classes”；在语法上，C 更是 C++ 的一个子集，C++ 几乎支持 C 语言的全部功能。如果采用 C++ 的方法来描述，以下方式恰如其分：

```
class C{};
class CPlusPlus : public C {};
```

C++ 继承自 C。

正是这种难以割舍的紧密联系使得 C/C++ 程序员必须对 C 有所重视。所以，本章就从 C++ 的前身——C 语言说起。

在开始这段学习旅程前，先分享一个只有程序员才明白的幽默：

有一次，她开玩笑似地问他：“我在你心里排第几？”他回头微笑着摸了摸她的头，用手指比划了个鸭蛋。她知道他在开玩笑，打了他一巴掌，尽管有些郁闷，但还是尽量避免流露出失望的神色。其实，因为她是文科生，所以她并不知道：在程序员的眼中，所有的数组、列表、容器的下标都是从 0 开始的。

所以，我们的建议也从 0 开始。

建议 0：不要让 main 函数返回 void

同 C 程序一样，每个 C++ 程序都包含一个或多个函数，而且必须有一个函数命名为 main，并且每个函数都由具有一定功能的语句序列组成。操作系统将 main 作为程序入口，调用 main 函数来执行程序；main 函数执行其语句序列，并返回一个值给操作系统。在大多数系统中，main 函数的返回值用于说明程序的退出状态。如果返回 0，则代表 main 函数成功执行完毕，程序正常退出，否则代表程序异常退出。

然而在编写 C++ 程序入口函数 main 的时候，很多程序员，特别是一些具有 C 基础的 C++ 程序员时经常会写出如下格式的 main 函数：

```
void main()
{
    // some code ...
}
```

上述代码在 VC++ 中是可以正确编译、链接、执行的。编译信息如下所示：

```
1>------ 已启动生成：项目：MainCpp，配置：Debug Win32 ------
1>  main.cpp
1>  MainCpp.vcxproj -> G:\MainCpp\Debug\MainCpp.exe
========== 生成：成功 1 个，失败 0 个，最新 0 个，跳过 0 个 ==========
```

但是当你将代码放在 Linux 环境下，采用 GCC 编译器进行编译时，你会吃惊地发现编译器抛出了如下的错误信息：

```
[develop@localhost ~]  g++ main.cpp
main.cpp:2: 错误：'::main' 必须返回 'int'
```

为什么同样的代码会出现两种不同的结果呢？这还是跨平台的 C/C++ 语言吗？不要对 C/C++ 的跨平台性产生质疑，之所以会这样，很大程度上要归结于市面上一些书的“误导”，以及微软对 VC++ 编译器 main 返回值问题的过分纵容。

在 C 和 C++ 中，不接收任何参数也不返回任何信息的函数原型为“void f(void);”。所以很多人认为，不需要程序返回值时可以把 main 函数定义成 void main(void)，然而这种想法是非常错误的！

有一点你必须明确：在 C/C++ 标准中从来没有定义过 void main() 这样的代码形式。C++ 之父 Bjarne Stroustrup 在他的主页 FAQ 中明确地写着这样一句话：

> 在 C++ 中绝对没有出现过 void main(){/* ... */} 这样的函数定义，在 C 语言中也是。

main 函数的返回值应该定义为 int 类型，在 C 和 C++ 标准中都是这样规定的。在 C99 标准中规定，只有以下两种定义方式是正确的[㊀]：

```
int main( void )
int main( int argc, char *argv[] )
```

在 C++03 中也给出了如下两种 main 函数的定义方式[㊁]：

```
int main()
int main( int argc, char *argv[] )
```

虽然在 C 和 C++ 标准中并不支持 void main()，但在部分编译器中 void main() 依旧是可以通过编译并执行的，比如微软的 VC++。由于微软产品的市场占有率与影响力很大，因此在某种程度上加剧了这种不良习惯的蔓延。不过，并非所有的编译器都支持 void main()，gcc 就站在了 VC++ 的对立面，它是这一不良习气的坚定抵制者，它会在编译时就明确地给出一个错误。

如果你坚持在某些编译器中使用 void main() 这种非标准形式的代码，那么当你把程序从

㊀ 参考资料：ISO/IEC 9899:1999 (E) Programming languages—C 5.1.2.2.1 Program startup。

㊁ 参考资料：ISO C++03 3.6.1 Main function。

一个编译器移植到另一个编译器时，你就要对可能出现的错误负责。

除了有 void main() 这样的不规范格式外，在 C 语言程序中，尤其是一些老版本的 C 代码中，你还会经常看到 main() 这样的代码形式。

一些老的 C 标准（诸如 C90）是支持 main() 这样的形式的。之所以支持，是因为在第一版的 C 语言中只有 int 一种数据类型，并不存在 char、long、float、double 等这些内置数据类型。既然只有 int 一种类型，也就不必显式地为 main 函数标明返回类型了。在 Brian W. Kernighan 和 Dennis M. Ritchie 的经典巨著《The C Programming Language，Second Edition》[⊖] 中用的就是 main()。后来，在 C 语言的改进版中数据类型得到了扩充，为了能兼容以前的代码，标准委员会就做出了如下规定：不明确标明返回值的，默认返回值为 int。在 C99 标准中，则要求编译器对于 main() 这种用法至少要抛出一个警告。

main 函数返回值的作用，可以采用下面的方法加以验证。

首先，编写 main.cpp 文件，文件内容如下所示：

```
int main()
{
   return 0;
}
```

在 Linux 环境下，采用命令：

```
g++ main.cpp
```

生成可执行文件 a.out。然后，执行命令：

```
./a.out && ehco "success"
```

结果输出 success。

修改上述程序：

```
int main()
{
   return -1;
}
```

做同样测试，无输出。

命令 A && B 中的 && 类似于 C++ 中的并操作（&&），如果 A 命令正确执行，接着就会执行命令 B；如果 A 出现异常，则 B 不执行。通过以上分析可知，当 main() 返回 0 时，a.out 正确执行并返回；但是如果返回 −1，程序就不能正常返回了。

最后，还得说明一下 C++ 标准中一个“好坏难定”的规定：

在 main 函数中，return 语句的作用在于离开 main 函数（析构掉所有具有动态生存时间

⊖ 本书已由机械工业出版社引进出版，中文书名为《C 程序设计语言（第 2 版 • 新版）》（ISBN 7-111-12806-0）和《C 程序设计语言（英文版 • 第 2 版）》（ISBN 7-111-19626-0）。——编辑注

的对象），并将其返回值作为参数来调用 exit 函数。如果函数执行到结尾而没有遇到 return 语句，其效果等同于执行了 return 0。⊖

也就是说，如果函数执行到 main 结束处时没有遇到 return 语句，编译器会隐式地为你加上 return 0;，效果与返回 0 相同。之所以说这条规定“好坏难定”，一方面是因为它让你省去了多敲几个字的麻烦；另一方面是因为这种便捷会让某些程序员忽视编译器代替他做的工作，而在思维中形成一种错误的认识：此函数可以无返回。

在应用这一规则时，你还得注意以下这两点：

- main 函数的返回类型是 int，不是 void 或其他类型。
- 该规则仅仅对 main 函数适用。

按照以上标准得到了一个完全合乎 C/C++ 标准的最小化的完整 C++ 程序：

```
int main()  {  }
```

本人不推荐使用上述这条规则，建议加上 return 0;，杜绝那些不必要误解。

请记住：

要想保证程序具有良好的可移植性能，就要标明 main 函数返回 int，而不是 void。强烈建议使用以下形式：

```
int main()
{
   // some processing codes
   return 0;
}
```

建议 1：区分 0 的 4 种面孔

0 在 C/C++ 语言中绝对是一个多面手，它扮演着多样的角色，拥有着多种面孔。总结起来包括以下几种角色：整型 0、空指针 NULL、字符串结束标志 '\0'、逻辑 FALSE/false，不同的角色适用于不同的情形，下面我们按照上述顺序一一介绍。

- 整型 0

这是我们最熟悉的一个角色。作为一个 int 类型，整型 0 占据 32 位的空间，其二进制表示为：

⊖ A return statement in main has the effect of leaving the main function (destroying any objects with automatic storage duration) and calling exit with the return value as the argument. If control reaches the end of main without encountering a return statement, the effect is that of executing return 0.—— ISO C++03 3.6.1 Main function.

```
00000000 00000000 00000000 00000000
```

它的使用方式最为简单直接，未经修饰，如下所示：

```
int nNum = 0;  // 赋值
if( nNum == 0 ) // 比较
```

❑ 空指针 NULL

NULL 是一个表示空指针常量的宏，在 C/C++ 标准中有如下阐述：

在文件 <clocale>、<cstddef>、<cstdio>、<cstdlib>、<cstring>、<ctime> 或者 <cwchar> 中定义的 NULL 宏，在国际标准中被认为是 C++ 空指针常量。⊖

指针与 int 类型所占空间是一样的，都是 32 位。那么，空指针 NULL 与 0 又有什么区别呢？还是让我们看一下 windef.h 中 NULL 的定义吧：

```
#ifndef NULL
#ifdef __cplusplus
#define NULL 0
#else
#define NULL ((void *)0)
#endif
#endif
```

通过定义我们可以看出，它们之间其实是没有太大的区别，只不过在 C 语言中定义 NULL 时会进行一次强制转型。我想之所以创造出 NULL，大概是为了增强代码的可读性，但这只是我的臆测，无从考究。

需要注意的是，这里的 0 与整型的 0 还是存在区别的。例如，int* pValue = 0; 是合法的，而 int* pValue = 1; 则是不合法的。这是因为 0 可以用来表示地址，但常数 1 绝对不行。

作为指针类型时，推荐按照下面的方式使用 0：

```
float* pNum = NULL;  // 赋值
if( pNum == NULL ) // 比较
```

❑ 字符串结束标志 '\0'

'\0' 与上述两种情形有所不同，它是一个字符。作为字符，它仅仅占 8 位，其二进制表示为：

```
00000000
```

因为字符类型中并没有与 0000 0000 对应的字符，所以就创造出了这么一个特殊字符。（对于类似 '\0' 这样的特殊字符，我们称之为转义字符。）在 C/C++ 中，'\0' 被作为字符串结束标志来使用，具有唯一性，与 '0' 是有区别的。

⊖ The macro NULL, defined in any of <clocale>, <cstddef>, <cstdio>, <cstdlib>, <cstring>,<ctime>, or <cwchar>, is an implementation-defined C + + null pointer constant in this International Standard.—— ISO C++03 C.2.2.3 Macro NULL

作为字符串结束符，0 的使用有些特殊。不必显式地为字符串赋值，但是必须明确字符串的大小。例如，在下面的代码中，“Hello C/C++”只有 11 个字符，却要分配 12 个字符的空间。

```
char sHello[12] = {"Hello C/C++"};  // 赋值
if( sHello[11] == '\0' )  // 比较
```

❑ 逻辑 FALSE/false

虽然将 FALSE/false 放在了一起，但是你必须清楚 FALSE 和 false 之间不只是大小写这么简单的差别。false/true 是标准 C++ 语言里新增的关键字，而 FALSE/TRUE 是通过 #define 定义的宏，用来解决程序在 C 与 C++ 环境中的差异。以下是 FALSE/TRUE 在 windef.h 中的定义：

```
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE 1
#endif
```

换言之，FALSE/TRUE 是 int 类型，而 false/true 是 bool 类型，两者是不一样的，只不过 C++ 帮我们完成了相关的隐式转换，以至于我们在使用中没有任何感觉。bool 在 C++ 里占用的是 1 个字节，所以 false 也只占用 1 个字节。

其二进制表示如下：

```
false -> 0
FALSE -> 00000000 00000000 00000000 00000000
```

在 C++ 中，推荐使用 bool 类型的 false/true，其使用方式如下：

```
bool isReady = false; // 赋值
if( isReady )   // 判断
```

如果不够细心，0 的多重性可能会让程序产生一些难以发现的 Bug，比如：

```
// 把 pSrc 指向的源字符串复制到 pDes 指向的内存块
while(pSrc)
{
   * pDes ++ = * pSrc ++;
}
```

正常情况下，当 pSrc 指向的字符为字符串结束符 '\0' 时，while 循环终止；但不幸的是，这里的条件写错了，while 终止条件变成了 pSrc 指向地址 0。结果 while 循环写入到内存中了，直至程序崩溃。

正确的写法应该是：

```
// 把 pSrc 指向的源字符串复制到 pDes 指向的内存块中
while(*pSrc)
{
```

```
    * pDes ++ = * pSrc ++;
}
```

请记住：

由于 0 存在多种面孔，容易让不细心的程序员产生混乱。唯一的解决办法就是在使用 0 的时候小心一点，再小心一点。

建议 2：避免那些由运算符引发的混乱

一般，C++ 被认为是 C 的超集。C++ 确实从它的前辈 C 那里继承了很多东西，比如一套含义相当混乱模糊的运算符。由于 C/C++ 语法规则的灵活性，以致那些粗心的程序员常会使用错误的运算符，进而引发不必要的麻烦。下面的代码就是一个典型的例子：

```
if(nValue = 0)
{
   // do something if nValue is not zero.
}
```

显然，程序员的本意是要写 if(nValue == 0)。不幸的是，上述语句虽未达成程序员的本意，但它却完全是合法的，编译器不会给出任何错误提示。C++ 语句首先会将 nValue 赋值为 0，然后再判断 nValue 是否为非零。结果就是 if 条件始终不能被满足，大括号中的代码永远不会被执行。

针对 = 和 == 之间的问题，通过良好的代码习惯可以避免，代码如下所示：

```
if(0 == nValue)
{
   // do something if nValue is not zero.
}
```

换句话说，就是将 0 和 nValue 的位置交换。此时，如果你再写出 if(0 = nValue) 这样的代码，编译器会直截了当地提示，发生了错误，编译失败。原因在于 0 = nValue 这样的代码在 C++ 语法中是不允许的，常数 0 不能作为左值来使用。

除了上述运算符，其他几对容易弄错的运算符是 &（按位与）和 &&（与），以及 |（按位或）和 ||（或）。对于这两对运算符，能够避免错误的只有细心。

请记住：

不要混淆 = 和 ==、& 和 &&、| 与 || 这三对运算符之间的差别，用细心和良好的代码习惯避免由于运算符混乱带来的麻烦。

建议 3：对表达式计算顺序不要想当然

一条一条的表达式构成了 C/C++ 代码的主体。接下来我们就来说说表达式的计算顺序。这些都是很琐碎的事情，但不可否认却又是非常有价值的。也许你会觉得下面的代码片段很熟悉：

```
if( nGrade & MASK == GRADE_ONE )
   ... // processing codes
```

很明显，当 grade 等于 GRADE_ONE 时 if 条件成立才是程序员的本意。可是上面的代码并没有正确地表达程序员的意思。这是因为位运算符（& 和 | ）的优先级低于关系运算符（比如 ==、<、>），所以上述代码的真实效果是：

```
if( nGrade & (MASK == GRADE_ONE) )
   ... // processing codes
```

这是很多人都容易犯的错误，我也有类似的经历，想当然地认为程序会按照设想的顺序来执行。这样的错误是很难发现的，调试起来也相当的费劲。C++/C 语言的运算符多达数十个，而这数十个运算符又具有不同的优先级与结合律，熟记它们确实比较困难，不过，可以用括号把意图表示得更清楚，所以不要吝啬使用括号，即使有时并不必要：

```
if( (nGrade & MASK) == GRADE_ONE )
   ... // processing codes
```

这样，代码就没有了歧义。

C/C++ 语言中存在着“相当险恶”的优先级问题，很多人很容易在这方面犯错误。如果代码表达式中包含较多的运算符，为了防止产生歧义并提高可读性，那么可以用括号确定表达式中每一个子表达式的计算顺序，不要过分自信地认为自己已经熟悉了所有运算符的优先级、结合律，多写几个括号确实是个好主意。例如：

```
COLOR rgb = (red<<16) | (green<<8) | blue;
bool isGradeOne = ((nGrade & MASK) == GRADE_ONE);
```

上面所说的计算顺序其实就是运算符的优先级，它只是一个“开胃菜”。接下来要说的是最为诡异的表达式评估求值的顺序问题。

因为 C++ 与 C 语言之间“剪不断理还乱”的特殊关系，C 语言中的好多问题也被带入到 C++ 的世界里了，包括表达式评估求值顺序问题。在 C 语言诞生之初，处理器寄存器是一种异常宝贵的资源；而复杂的表达式对寄存器的要求很高，这使得编译器承受着很大的压力。为了能够使编译器生成高度优化的可执行代码，C 语言创造者们就赋予了寄存器分配器这种额外的能力，使得它在表达式如何评估求值的问题上留有很大的处理余地。虽然当今寄存器有了极大的进步，对复杂表达式的求值不再有什么压力，但是赋予寄存器分配器的这种能力

却一直没有收回，所以在 C++ 中评估求值顺序的不确定性仍然存在，而且这很大程度上决定于你所使用的编译器。这就要求软件工程师更加认真仔细，以防对表达式设定了无依据、先入为主的主观评估顺序。

这其实也是 C 语言的陷阱之一，《The C Programming Lauguage》（程序员亲切地称此书为“K & R”）中反复强调，函数参数也好，某个操作符中的操作数也罢，表达式求值次序是不一定的，每个特定机器、操作系统、编译器也都不一样。就像《The C Programming Language》影印版第 2 版的 52 页所说的那样：

如同大多数语言一样，C 语言也不能识别操作符的哪一个操作数先被计算（&&、||、?: 和，四种操作符除外），例如 x=f()+g()。⊖

这里所说的求值顺序主要包括以下两个方面：

❑ 函数参数的评估求值顺序

分析下面代码片段的输出结果：

```
int i = 2010;
printf("The results are: %d %d", i, i+=1 );
```

函数参数的评估求值并没有固定的顺序，所以，printf() 函数的输出结果可能是 2010、2011，也可能是 2011、2011 。

类似的还有：

```
printf("The results are: %d %d", p(), q() );
```

p() 和 q() 到底谁先被调用，这是一个只有编译器才知道的问题。

为了避免这一问题的发生，有经验的工程师会保证凡是在参数表中出现过一次以上的变量，在传递时不改变其值。即使如此也并非万无一失，如果不是足够小心，错误的引用同样会使努力前功尽弃，如下所示：

```
int para = 10;
int &rPara = para;
int f(int, int);
int result = f(para, rPara *= 2);
```

推荐的形式应该是：

```
int i = 2010;
printf("The results are: %d %d", i, i+1 );

int para = p();
printf("The results are: %d %d", para, q() );
```

⊖ （C, like most languages, does not specify the order in which the operands of an operator are evaluated. (exceptions are && || ?: and ",") for example x = f() + g();）——K&R Second Edition P52

```
int para = 10;
int f(int, int);
int result = f(para, para*2);
```

❑ 操作数的评估求值顺序

操作数的评估求值顺序也不固定，如下面的代码所示：

```
a = p() + q() * r();
```

三个函数 p()、q() 和 r() 可能以 6 种顺序中的任何一种被评估求值。乘法运算符的高优先级只能保证 q() 和 r() 的返回值首先相乘，然后再加到 p() 的返回值上。所以，就算加上再多的括号依旧不能解决问题。

幸运的是，使用显式的、手工指定的中间变量可以解决这一问题，从而保证固定的子表达式评估求值顺序：

```
int para1 = p();
int para2 = q();
a = para1 + para2 * r();
```

这样，上述代码就为 p()、q() 和 r() 三个函数指定了唯一的计算顺序：p() → q() → r()。

另外，有一些运算符自诞生之日起便有了明确的操作数评估顺序，有着与众不同的可靠性。例如下面的表达式：

```
(a < b) && (c < d)
```

C/C++ 语言规定，a < b 首先被求值，如果 a < b 成立，c < d 则紧接着被求值，以计算整个表达式的值。但如果 a 大于或等于 b，则 c < d 根本不会被求值。类似的还有 ||。这两个运算符的短路算法特性可以让我们有机会以一种简约的、符合习惯用法的方式表达出很复杂的条件逻辑。

三目条件运算符 ?: 也起到了把参数的评估求值次序固定下来的作用：

```
expr1 ? expr2 : expr3
```

第一个表达式会首先被评估求值，然后第二个和第三个表达式中的一个会被选中并评估求值，被选中并评估求值的表达式所求得的结果就会作为整个条件表达式的值。

此外，在建议 6 中将会详细介绍的逗号运算符也有固定的评估求值顺序。

请记住：

表达式计算顺序是一个很繁琐但是很有必要的话题：

❑ 针对操作符优先级，建议多写几个括号，把你的意图表达得更清晰。

❑ 注意函数参数和操作数的评估求值顺序问题，小心其陷阱，让你的表达式不要依赖计算顺序。

建议 4：小心宏 #define 使用中的陷阱

C 语言宏因为缺少必要的类型检查，通常被 C++ 程序员认为是“万恶之首”，但就像硬币的两面一样，任何事物都是利与弊的矛盾混合体，宏也不例外。宏的强大作用在于在编译期自动地为我们产生代码。如果说模板可以通过类型替换来为我们产生类型层面上多样的代码，那么宏就可以通过符号替换在符号层面上产生的多样代码。正确合理地使用宏，可以有效地提高代码的可读性，减少代码的维护成本。

不过，宏的使用中存在着诸多的陷阱，如果不注意，宏就有可能真的变成 C++ 代码的“万恶之首”。

（1）用宏定义表达式时，要使用完备的括号。

由于宏只是简单的字符替换，宏的参数如果是复合结构，那么替换之后要是不用括号保护各个宏参数，可能会由于各个参数之间的操作符优先级高于单个参数内部各部分之间相互作用的操作符优先级，而产生意想不到的情形。但并不是使用了括号就一定能避免出错，我们需要完备的括号去完备地保护宏参数。

如下代码片段所定义的宏要实现参数 a 和参数 b 的求和，但是这三种定义都存在一定风险：

```
#define ADD( a, b )  a + b
#define ADD( a, b )  (a + b)
#define ADD( a, b )  (a) + (b)
```

例如，ADD(a,b) * ADD(c,d) 的本意是对 (a+b)*(c+d) 求值，在采用了上面定义的宏之后，代码展开却变成了如下形式，其中只有第 2 种方式“碰巧”实现了原本意图：

```
a + b * c + d
(a + b) * (c + d)
(a) + (b) * (c) + (d)
```

之所以说“碰巧”，是因为第 2 种方式中括号的使用也非完备的。例如：

```
#define MULTIPLE( a, b )  (a * b)
```

在计算 (a+b)×c 时，如果采用上述宏 MULTIPLE（a+b，c），代码展开后，我们得到的却是 a+b×c 的结果。

要避免这些问题，要做的就是：用完备的括号完备地保护各个宏参数。正确的定义应为：

```
#define ADD( a, b )  ((a)+(b))
#define MULTIPLE( a, b )  ((a)*(b))
```

（2）使用宏时，不允许参数发生变化。

宏参数始终是一个比较敏感、容易引发错误的东西。有很多人认为，在某种程度上带参的宏定义与函数有几分类似。但是必须注意它们的区别，正如下面代码片段所示：

```
#define SQUARE( a ) ((a) * (a))
int Square(int a)
{
    return a*a;
}

int nValue1 = 10, nValue2 = 10;
int nSquare1 = SQUARE(nValue1++); // nSquare1=110, nValue1=12
int nSquare2 = Square(nValue2++);// nSquare2=100, nValue2=11
```

类似的定义，却产生了不同的结果，究其原因还是宏的字符替换问题。正如上面的示例一样，两处的 a 都被参数 nValue1++ 替换了，所以 nValue1 自增操作也就被执行了两回。

这就是宏在展开时对其参数的多次取值替换所带来的副作用。为了避免出现这样的副作用，最简单有效的方法就是保证宏参数不发生变化，如下所示。

```
#define SQUARE( a ) ((a) * (a))

int nValue1 = 10;
int nSquare1 = SQUARE(nValue1); // nSquare1=100
nValue1++; //  nValue1=11
```

（3）用大括号将宏所定义的多条表达式括起来。

如果宏定义包含多条表达式，一定要用大括号将其括起来。如果没有这个大括号，宏定义中的多条表达式很有可能只有第一句会被执行，正如下面的代码片段：

```
#define CLEAR_CUBE_VALUE( l, w, h )\
    l = 0;\
    w = 0;\
    h = 0;

int i = 0;
for (i = 0; i < CUBE_ACOUNT; i++)
    CLEAR_CUBE_VALUE( Cubes[i].l, Cubes[i].w, Cubes[i].h );
```

简单的字符替代，并不能保证多条表达式都会放入 for 循环的循环体内，因为没有将它包围在循环体内的大括号中。正确的做法应该是用大括号将多条表达式括起来，这样就能保证多条表达式全部执行了，如下面的代码片段所示：

```
#define CLEAR_CUBE_VALUE( l, w, h )\
{\
    l = 0;\
    w = 0;\
    h = 0;\
}
```

请记住：

正确合理使用 C 语言中的宏，能有效地增强代码的可读性。但是也要遵守一定的规则，

避免踏入其中的陷阱：（1）用宏定义表达式时，要使用完备的括号。（2）使用宏时，不允许参数发生变化。（3）用大括号将宏所定义的多条表达式包括起来。

建议 5：不要忘记指针变量的初始化

可以说指针是 C/C++ 语言编程中最给力的工具。指针，让我们直接去面对最为神秘的内存空间，赋予我们对内存进行直接操作的能力。由于指针操作执行速度快、占用内存少，众多程序员对它深爱不已。但是，它的灵活性和难控制性也让许多程序员觉得难以驾驭，以致到了谈指针色变的程度。

指针就是一把双刃剑。用好了它，会给你带来诸多便利，反之，则往往会引发意想不到的问题。

其中，指针的初始化就是我们应当重视的问题之一。指针应当被初始化，这是一个毋庸置疑的问题，关键是应该由谁来负责初始化，是编译器，还是程序员自己？

为了更好地贯彻零开销原则（C++ 之父 Bjarne 在设计 C++ 语言时所遵循的原则之一，即“无须为未使用的东西付出代价”），编译器一般不会对一般变量进行初始化，当然也包括指针。所以负责初始化指针变量的只有程序员自己。

使用未初始化的指针是相当危险的。因为指针直接指向内存空间，所以程序员很容易通过未初始化的指针改写该指针随机指向的存储区域。而由此产生的后果却是不确定的，这完全取决于程序员的运气。例如下面的程序片段：

```
#include <iostream>
int main()
{
    int *pInt;
    std::cout<<pInt<<"\n";
    return 0;
}
```

在 VC++ 中，程序在 Release 模式下输出 0x004080d0，而在 Debug 模式下输出 0xcccccccc。很明显未初始化的指针指向的是一个随机的地址。如果对其执行写操作会怎样？那很有可能会直接导致程序崩溃。

可以将指针初始化为某个变量的地址。需要注意的是，当用另一个变量的地址初始化指针变量时，必须在声明指针之前声明过该变量。代码片段如下所示：

```
int number = 0;     // Initialized integer variable
int* pNumber = &number;     // Initialized pointer
```

当然，我们在必要时也可以将其初始化为空指针 0（NULL）：

```
int* pNumber = NULL;     // Initialized pointer as NULL
```

如果使用未初始化的局部变量，程序编译时会给出警告 C4700：

```
warning C4700: 使用了未初始化的局部变量 "**"
```

需要注意警告中的四个字“局部变量”。因为对于全局变量来说，在声明的同时，编译器会悄悄完成对变量的初始化。代码片段如下所示：

```
#include <iostream>
int *pInt;
int main()
{
   std::cout<<pInt<<"\n";
   return 0;
}
```

此时，程序编译不会再出现警告，程序输出：00000000。

请记住：

使用未初始化的局部指针变量是件很危险的事，所以，在使用局部指针变量时，一定要及时将其初始化。

建议 6：明晰逗号分隔表达式的奇怪之处

逗号分隔的表达式是从 C 继承而来的。它用一种特殊的运算符——逗号运算符将多个表达式连接起来。逗号表达式的一般形式为：

```
表达式 1，表达式 2，表达式 3..... 表达式 n
```

需要注意的是，整个逗号分隔表达式的值为表达式 n 的值。

在使用 for- 循环和 while- 循环时，经常会使用这样的表达式。然而，由于语言规则不直观，因此理解这样的语句存在一定的困难。例如：

```
if(++x, --y, x<20 && y>0)  /* 三个表达式 */
```

if 条件包含由逗号分隔的三个表达式。C++ 确保每个表达式都会被执行，并产生作用。不过，整个表达式的值仅是最右边表达式的结果。因此，只有当 x 小于 20 且 y 大于 0 时才会返回 true，上述条件也才会为真。再举一个逗号表达式的例子：

```
int j=10;
int i=0;
while( ++i, --j)
{
```

```
    /* 只要 j 不为 0 就会循环执行 */
}
```

其实，逗号表达式无非是把若干个表达式 " 串联 " 起来。在许多情况下，使用逗号表达式的目的只是想分别得到各个表达式的值，而并非一定需要得到和使用整个逗号表达式的值。

当然并不是所有地方出现的逗号都是逗号运算符，例如用逗号分隔的函数参数：

```
printf("%d - %s - %f", count, str, PI);
```

“count, str, PI”并非逗号分隔表达式，而是 printf 的三个输入参数。

另外一个需要注意的问题就是，在 C++ 中，逗号分隔表达式既可以用作左值，又可以用作右值。

请记住：

逗号分隔的表达式由于语言规则的不直观，容易产生理解上的误差。在使用逗号分隔表达式时，C++ 会确保每个表达式都被执行，而整个表达式的值则是最右边表达式的结果。

建议 7：时刻提防内存溢出

作为一个程序员，对内存溢出问题肯定不陌生，它已经是软件开发历史上存在了近 40 年的大难题。在内存空间中，当要表示的数据超出了计算机为该数据分配的空间范围时，就产生了溢出，而溢出的多余数据则可以作为指令在计算机中大摇大摆地运行。不幸的是，一不小心这就成了黑客们可利用的秘密后门，“红色代码”病毒事件就是黑客利用内存溢出攻击企业网络的“经典案例”。甚至有人称，操作系统中超过 50% 的安全漏洞都是由内存溢出引起的。

众所周知，C/C++ 语言虽然是一种高级语言，但是其程序的目标代码却非常接近机器内核，它能够直接访问内存和寄存器，这种特性大大提升了 C/C++ 语言代码的性能，同时也提高了内存溢出问题出现的可能性。内存溢出问题可以说是 C/C++ 语言所固有的缺陷，因为它们既不检查数组边界，也不检查类型可靠性。

假设代码申请了 X 字节大小的内存缓冲区，随后又向其中复制超过 X 字节的数据，那么多出来的字节会溢出原本的分配区。最重要的是，C/C++ 编译器开辟的内存缓冲区常常邻近重要的数据结构。如果恶意攻击者用“别有用心”的东西刻意地覆盖原本安全可信的数据，那么后果就是此机器将会成为他们肆意攻击的“肉鸡”。下面将介绍常见的缓冲区溢出，以及预防措施。

C 语言中的字符串库没有采用相应的安全保护措施，所以在使用时要特别小心。例如，在执行 strcpy、strcat 等函数操作时没有检查缓冲区大小，就会很容易引起安全问题。

现在分析下面的代码片段：

```
const int MAX_DATA_LENGTH = 32;
void DataCopy (char *szSrcData)
 {
   char szDestData[MAX_DATA_LENGTH];
   strcpy(cDest,szData);
   // processing codes
   ...
  }
```

似乎这段代码不存在什么问题，但是细心的读者还是会发其中的危险。如果数据源 szSrcDatar 的长度不超过规定的长度，那么这段代码确实没什么问题。strcpy() 不会在乎数据来源，也不会检查字符串长度，唯一能让它停下来的只有字符串结束符 '\0'。不过，如果没有遇到这个结束符，它就会一个字节一个字节地复制 szSrcData 的内容，在填满 32 字节的预设空间后，溢出的字符就会取代缓冲区后面的数据。如果这些溢出的数据恰好覆盖了后面 DataCopy 函数的返回地址，在该函数调用完毕后，程序就会转入攻击者设定的“返回地址”中，乖乖地进入预先设定好的陷阱。

为了避免落入这样的圈套，给作恶者留下可乘之机，当 C/C++ 代码处理来自用户的数据时，应该处处留意。如果一个函数的数据来源不可靠，又要用到内存缓冲区，那么必须提高警惕，必须知道内存缓冲区的总长度，并检验内存缓冲区。

```
const int MAX_DATA_LENGTH = 32;
void DataCopy (char *szSrcData, DWORD nDataLen)
 {
   char szDestData[MAX_DATA_LENGTH];
   if(nDataLen < MAX_DATA_LENGTH)
      strcpy(cDest,szData);
   szDestData[nDataLen] = '\0'; // 0x42;
   // processing code
   ...
  }
```

首先，要获得 szSrcData 的长度，保证数据长度不大于最大缓冲区长度 MAX_DATA_LENGTH；其次，要保证参数传来的数据长度真实有效，方法就是向内存缓冲区的末尾写入数据。因为，当缓冲区溢出时，一旦向其中写入常量值，代码就会出错，终止运行。与其落入阴谋家的陷阱，还不如及时终止程序运行。

虽然上述方法能够有效地降低内存溢出的危害，却不能从根本上避免对内存溢出的攻击。所以在调用 strcpy、strcat、gets 等经典函数时，你要从源代码开始就提高警惕，尽量追踪传入数据的流向，向代码中的每一个假设提出质疑，包括对那些所谓相对安全可靠的改良

版 N-Versions（strncpy 或 strncat）也不可轻信。

访问边界数据同样可能引起缓冲区溢出。在这种情况下的内存溢出不会像第一种那么危险，但同样令人讨厌。就如下面的代码片段：

```
const int DATA_LENGTH = 16;
Int data[16] = {1,9,8,4,0,9,1,7,1,9,8,7,0,3,0,9};
void PrintData()
{
    for(int i=0; a[i]!=0&&i<DATA_LENGTH; i++)
    {
         cout<<data[i])<<endl;
     }
}
```

这也是一个隐藏很深、难以发现的问题：当 i==16 的时候，在判断 i< DATA_LENGTH 的同时需要判断 data[16]。而 data[16] 已经访问到了非法区域，可能引起缓冲区溢出。正确的方式应该是不要将索引号 i 与数据本身 data[i] 的判断放在一起，而是将判断条件分成两句：

```
const int DATA_LENGTH = 16;
int data[16] = {1,9,8,4,0,9,1,7,1,9,8,7,0,3,0,9};
void PrintData()
{
    for(int i=0; i<DATA_LENGTH; i++)
    {
        if(a[i]!=0)
           cout<<data[i])<<endl;
    }
}
```

类似的问题还有可能发生在访问未初始化指针或失效指针时。未初始化的指针和失效后未置 NULL 的指针指向的是未知的内存空间，所以对这样的指针进行操作很有可能访问或改写未知的内存区域，也就可能引起缓冲区溢出的问题了。

请记住：

因为内存溢出潜在的危害很大，所以必须注意和面对这个问题，特别是在网络相关的应用程序中。在调用 C 语言字符串经典函数（如 strcpy、strcat、gets 等）时，要从源代码开始就提高警惕，尽量追踪传入数据的流向，向代码中的每一个假设提出质疑。在访问数据时，注意对于边界数据要特殊情况特殊处理，还要对杜绝使用未初始化指针和失效后未置 NULL 的“野指针”。

建议 8：拒绝晦涩难懂的函数指针

在 C/C++ 程序中，数据指针是最直接也是最常用的，理解起来也相对简单容易，但是函数指针理解起来却并不轻松。函数指针在运行时的动态调用中应用广泛，是一种常见而有效的手段。但是，如果不注重一定的使用技巧，函数指针也会变得晦涩难懂。

告诉我下面定义的含义是什么？

```
void (*p[10]) (void (*)());
```

如此繁琐的语法定义几乎难以辨认，这与我们提倡的可读性背道而驰了。这样的函数指针之所以让程序员发愁，最主要的原因是它的括号太多了，往往会让程序员陷在括号堆中理不清头绪。下面一层一层地来分析吧。第一个括号中的 *p[10] 是一个指针数组，数组中的指针指向的是一些函数，这些函数参数为 void (*)()，返回值为空；参数部分的 void (*)() 是一个无参数、返回值为空的函数指针。

分析这样的代码简直是一种折磨。如何有效地提高函数指针定义的可读性呢？那就是使用 typedef。typedef 方法可以有效地减少括号的数量，可以通过 typedef 来合理地简化这些声明，理清层次，所以它的使用倍受推荐。

以上面的定义为例。首先，声明一个无参数、返回空的函数指针的 typedef，如下所示：

```
typedef void (*pfv)();
```

接下来，声明另一个 typedef，一个指向参数为 pfv 且返回为空的函数指针：

```
typedef void (*pFun_taking_pfv) (pfv);
```

现在，再去声明一个含有 10 个这样指针的数组就变得轻而易举了，而且可读性有了很大的提升：

```
pFun_taking_pfv p[10]; /* 等同于 void (*p[10]) (void (*)());*/
```

请记住：

函数指针在运行时的动态调用（例如函数回调）中应用广泛。但是直接定义复杂的函数指针会由于有太多的括号而使代码的可读性下降。使用 typedef 可以让函数指针更直观和易维护。拒绝晦涩难懂的函数指针定义，拒绝函数定义中成堆的括号。

建议 9：防止重复包含头文件

假设，我们的工程中有如下三个文件：a.h、b.h 和 c.cpp，其中 b 文件中包含了 a.h，c 文件中又分别包含了 a.h 和 b.h 两个文件，如图 1-1 所示。

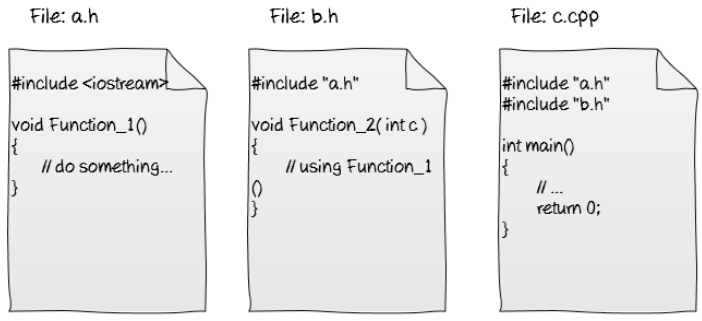

图 1-1 工程文件示例

在编译整个工程时，编译器会出现“multiple definition of”错误。原因在于 a.h 文件被包含了两次。为了避免同一个文件被包含多次，C/C++ 中有两种处理方式，一种是 #ifndef 方式，另一种是 #pragma once 方式。

方式 1：

```
#ifndef __SOMEFILE_H__
#define __SOMEFILE_H__
... ... // 声明、定义语句
#endif
```

方式 2：

```
#pragma once
... ... // 声明、定义语句
```

C/C++ 语言标准支持第一种方式。这种方式不仅可以保证同一个文件不会被包含多次，也能保证内容完全相同的两个文件不会被同时包含。当然，其缺点就是如果一不小心在不同头文件中定义了相同的宏名，造成了宏名“撞车”，那就可能会导致明明看到头文件存在，编译器却硬说找不到声明，这确实会令人非常恼火。为了避免宏名“撞车”，保证宏的唯一性，建议按照 Google 公司建议的那样，头文件基于其所在项目源代码树的全路径而命名。命名格式为：

```
<PROJECT>_<PATH>_<FILE>_H_
```

由于编译器在每次编译时都需要打开头文件才能判定是否有重复定义，因此在编译大型项目时，ifndef 会使编译时间相对较长。

#pragma once 方式一般由编译器提供，它保证同一个文件不会被包含多次。这里所说的“同一个文件”指的是物理上的一个文件，而不是指内容相同的两个文件。#pragma once 声明只针对文件，而不能针对某一文件中的一段代码。这种方式避免了因想方设法定义一个独一无二的宏而产生的烦恼；另外，针对大型项目的编译速度也有了提升。但是这种方式因为不受 C/C++ 语言标准支持，所以受到了编译器的限制，它在兼容性方面表现得不是很好。因

此很多程序员为了代码的兼容性，宁肯降低一些编译性能，而选择遵循 C/C++ 标准，采用第一种方式。

注意　针对 #pragma once，GCC 已经取消了对其的支持，而微软的 VC++ 却仍在坚持。

请记住：

为了避免重复包含头文件，建议在声明每个头文件时采用“头文件卫士”加以保护，比如采用如下的形式：

```
#ifndef _PROJECT_PATH_FILE_H_
#define _PROJECT_PATH_FILE_H_
... ... // 声明、定义语句
#endif
```

建议 10：优化结构体中元素的布局

下面的代码片段定义了结构体 A 和 B：

```
struct A  // 结构体 A
{
     int a;
     char b;
     short c;
};
struct B  // 结构体 B
{
     char b;
     int a;
     short c;
};
```

在 32 位机器上，char、short、int 三种类型的大小分别是 1、2、4。那么上面两个结构体的大小如何呢？

结构体 A 中包含了一个 4 字节的 int，一个 1 字节的 char 和一个 2 字节的 short，B 也一样，所以 A、B 的大小应该都是 4+2+1 = 7 字节。但是，实验给出的却是另外的结果：

```
sizeof(strcut A) = 8, sizeof(struct B) = 12
```

其原因还要从字节对齐说起。

现代计算机中内存空间都是按照字节来划分的，从理论上来讲，对变量的访问可以从任何地址开始；但在实际情况中，为了提升存取效率，各类型数据需要按照一定的规则在空间上排列，这使得对某些特定类型的数据只能从某些特定地址开始存取，以空间换取时间，这

就是字节对齐。

结构体默认的字节对齐一般满足三个准则：

（1）结构体变量的首地址能够被其最宽基本类型成员的大小所整除。

（2）结构体每个成员相对于结构体首地址的偏移量（offset）都是成员自身大小的整数倍，如有需要，编译器会在成员之间加上填充字节（Internal Adding）。

（3）结构体的总大小为结构体最宽基本类型成员大小的整数倍，如有需要，编译器会在最末一个成员之后加上填充字节（Trailing Padding）。

按照这三条规则再去分析结构体 A 和 B，就不会对于上述的结果一脸诧异了。这两个结构体在内存空间中的排列如图 1-2 所示（灰色网格表示的字节为填充字节）。

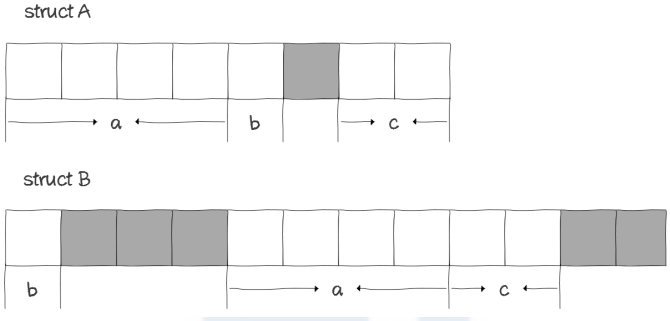

图 1-2 结构体 A 和 B 的内存分布

在编程应用中，如果空间紧张，需要考虑节约空间，那么就需要将结构体中的各个变量按照上面的原则进行排列。基本的原则是：把结构体中的变量按照类型大小从小到大依次声明，尽量减少中间的填充字节。

也可以采用保留字节的形式显式地进行字节填充实现对齐，以提高存取效率。其实这就是时间与空间的博弈。如下面的代码片段所示，其中的 reserved 成员对程序没有什么意义，它只是填补空间以达到字节对齐的目的：

```
struct A  // 结构体 A
{
   int a;
   char b;
   char reserved; // 保留字节，空间换时间
   short c;
};
struct B  // 结构体 B
{
   char b;
   char reserved1[3]; // 保留字节 1，空间换时间
   int a;
   short c;
```

```
    char reserved2[2]; // 保留字节 2，空间换时间
};
```

在某些时候，还可以通过编译器的 pack 指令调整结构体的对齐方式。#pragma pack 的基本用法为：

```
#pragma pack( n )
```

n 为字节对齐数，其取值为 1、2、4、8、16，默认是 8。

```
#pragma pack(1)// 设置 1 字节对齐
struct A  // 结构体 A
{
    int a;
    char b;
    short c;
};
```

将结构体 A 的对齐方式设为 1 字节对齐，那么 A 就不再有填充字节了，sizeof(A) 的结果即为各元素所占字节之和 7。

请记住：

了解结构体中元素的对齐规则，合理地为结构体元素进行布局。这样不仅可以有效地节约空间，还可以提高元素的存取效率。

建议 11：将强制转型减到最少

C++ 在设计中一直强调类型安全，而且也采取了一定的措施来保障这条准则的执行。但是，从 C 继承而来的强制转型却破坏了 C++ 类型系统，C 中的强制转型可谓是“无所不能”，其超强的能力给 C++ 带来了很大的安全隐患。强制转型会引起各种各样的麻烦，有时这些麻烦很容易被察觉，有时它们却又隐藏极深，难以察觉。

在 C/C++ 语言中，强制转型是“一个你必须全神贯注才能正确使用”的特性。所以一定要慎用强制转型。

首先来回顾一下 C 风格（C-style）的强制转型语法，如下所示：

```
// 将表达式的类型转换为 T
(T) expression
 T(expression)
```

这两种形式之间没有本质上的区别。在 C++ 中一般称为旧风格的强制转型。

在赋值时，强制类型的转换形式会让人觉得不精密、不严格，缺乏安全感，主要是因为不管表达式的值是什么类型，系统都自动将其转为赋值运算符左侧变量的类型。而转变后数

据可能会有所不同，若不加注意，就可能产生错误。

将较大的整数转换为较短的数据类型时，会产生无意义的结果，而程序员可能被蒙在鼓里。正如下面的代码片段所示：

```
unsigned i = 65535;
int j = (int) i;
```

输出结果竟然成了－1。较长的无符号类型在转换为较短的有符号类型时，其数值很可能会超出较短类型的数值表示范围。编译器不会监测这样的错误，它所能做的仅仅是抛出一条非安全类型转换的警告信息。如果这样的问题发生在运行时，那么一切会悄无声息，系统既不会中断，也不会出现任何的出错信息。

类似的问题还会发生在有符号负数转化为无符号数、双精度类型转化为单精度类型、浮点数转化为整型等时候。以上这些情况都属于数值的强制转型，在转换过程中，首先生成临时变量，然后会进行数值截断。

在标准 C 中，强制转型还有可能导致内存扩张与截断。这是因为在标准 C 中，任何非 void 类型的指针都可以和 void 类型的指针相互指派，也就可以通过 void 类型指针这个中介，实现不同类型的指针间接相互转换了。代码如下所示：

```
double PI = 3.1415926;
double *pd = &PI;
void *temp = pd;
int *pi = temp; // 转换成功
```

指针 pd 指向的空间本是一个双精度数据，8 字节。但是经过转换后，pi 却指向了一个 4 字节的 int 类型。这种发生内存截断的设计缺陷会在转换后进行内存访问时存在安全隐患。不过，这种情况只会发生在标准 C 中。在 C++ 中，设计者为了杜绝这种错误的出现，规定了不同类型的指针之间不能相互转换，所以在使用纯 C++ 编程时大可放心。而如果 C++ 中嵌入了部分 C 代码，就要注意因强制转型而带来的内存扩张或截断了。

与旧风格的强制转型相对应的就是新风格的强制转型了，在 C++ 提供了如下四种形式：

```
const_cast(expression)
dynamic_cast(expression)
reinterpret_cast(expression)
static_cast(expression)
```

新风格的强制转型针对特定的目的进行了特别的设计，如下所示。

❑ const_cast<T*> (a)

它用于从一个类中去除以下这些属性：const、volatile 和 __unaligned。

```
class A  {  // …  };
void Function()
{
```

```
    const A *pConstObj = new A;
    A *pObj = pConstObj; //ERROR: 不能将 const 对象指针赋值给非 const 对象
    pObj = const_cast<A*>( pConstObj); // OK
    //...
}
```

这种强制转型的目的简单明确，使用情形比较单一，易于掌握。

❑ dynamic_cast<T*>(a)

它将 a 值转换成类型为 T 的对象指针，主要用来实现类层次结构的提升，在很多书中它被称做“安全的向下转型（Safe Downcasting）”，用于继承体系中的向下转型，将基类指针转换为派生类指针，这种转换较为严格和安全。如下面的代码片段所示：

```
class B {  //...  };
class D : public B {  //...  };
void Function(D *pObjD)
{
    D *pObj = dynamic_cast<D*>( pObjD);
    //...
}
```

如果 pObjD 指向一个 D 类型的对象，pObj 则指向该对象，所以对该指针执行 D 类型的任何操作都是安全的。但是，如果 pObjD 指向的是一个 B 类型的对象，pObj 将是一个空指针，这在一定程度上保证了程序员所需要的“安全”，只是，它也付出了一定的运行时代价，而且代价非常大，实现相当慢。有一种通用实现是通过对类名称进行字符串比较来实现的，只是其在继承体系中所处的位置越深，对 strcmp 的调用就越多，代价也就越大。如果应用对性能要求较高，那么请放弃 dynamic_cast。

❑ reinterpret_cast<T*>(a)

它能够用于诸如 One_class* 到 Unrelated_class* 这样的不相关类型之间的转换，因此它是不安全的。其与 C 风格的强制转型很是相似。

```
class A { // ... };
class B { //... };
void f()
{
   A* pa = new A;
   B* pb = reinterpret_cast<B*>(pa);
   // ...
}
```

在不了解 A、B 内存布局的情况下，强行将其进行转换，很有可能出现内存膨胀或截断。

❑ static_cast<T*>(a)

它将 a 的值转换为模板中指定的类型 T。但是，在运行时转换过程中，它不会进行类型检查，不能确保转换的安全性。如下面的代码片段所示：

```
class B { ... };
class D : public B { ... };
void Function(B* pb, D* pd)
{
     D* pd2 = static_cast<D*>(pb);    // 不安全
     B* pb2 = static_cast<B*>(pd);    // 安全的
 }
```

之所以说第一种是不安全的，是因为如果 pb 指向的仅仅是一个基类 B 的对象，那么就会凭空生成继承信息。至于这些信息是什么、正确与否，无从得知。所以对它进行 D 类型的操作将是不安全的。

C++ 是一种强类型的编程语言，其规则设计为“保证不会发生类型错误”。在理论层面上，如果希望程序顺利地通过编译，就不应该试图对任何对象做任何不安全的操作。不幸的是，继承自 C 语言的强制转型破坏了类型系统，所以建议尽量少地使用强制转型，无论是旧的 C 风格的还是新的 C++ 风格的。如果发现自己使用了强制转型，那么一定要小心，这可能就是程序出现问题的一个信号。

请记住：

由于强制转型无所不能，会给 C++ 程序带来很大的安全隐患，因此建议在 C++ 代码中，努力将强制转型减到最少。

建议 12：优先使用前缀操作符

也许从开始接触 C/C++ 程序的那天起，就记住了前缀和后缀运算，知道了 ++ 的前缀形式是“先加再用”，后缀形式是“先用再加”。前缀和后缀运算是 C 和 C++ 语言中的基本运算，它们具有类似的功能，区别也很细微，主要体现在运行效率上。

分析下面的代码片段：

```
int n=0, m=0;
n = ++m; /*m 先加 1，之后赋给 n*/
cout << n << m; /* 结果：1 1*/
```

在这个例子中，赋值之后，n 等于 1，因为它是在将 m 赋予 n 之前完成的自增操作。再看下面的代码：

```
int n=0, m=0;
n = m++; /* 先将 m 赋于 n，之后 m 加 1*/
cout << n << m; /* 结果：0 1*/
```

这个例子中，赋值之后，n 等于 0，因为它是先将 m 赋予 n，之后 m 再加 1 的。

为了更好地理解前缀操作符和后缀操作符之间的区别，可以查看这些操作的反汇编代

码。即使不了解汇编语言，也可以很清楚地看到二者之间的区别，注意 inc 指令出现的位置：

```
/* m=n++; 的反汇编代码 */
mov ecx, [ebp-0x04] /*store n's value in ecx register*/
mov [ebp-0x08], ecx /*assign value in ecx to m*/
inc dword ptr [ebp-0x04] /*increment n*/
/*m=++n; 的反汇编代码 */
inc dword ptr [ebp-0x04] /*increment n;*/
mov eax, [ebp-0x04] /*store n's value in eax register*/
mov [ebp-0x08], eax /*assign value in eax to m*/
```

从汇编代码可以看出，两者采取了相同的操作，只是顺序稍有不同而已。但是，前缀操作符的效率要优于后缀操作符，这是因为在运行操作符之前编译器需要建立一个临时的对象，而这还要从函数重载说起。

重载函数间的区别取决于它们在参数类型上的差异，但不论是自增的前缀还是后缀，都只有一个参数。为了解决这个语言问题，C++ 规定后缀形式有一个 int 类型的参数，当函数被调用时，编译器传递一个 0 作为 int 类型参数的值给该函数：

```
// 成员函数形式的重载
< Type > ClassName :: operator ++ ( ); // 前缀
< Type > ClassName :: operator ++ ( int ); // 后缀
// 非成员函数形式的重载
< Type > operator ++ (ClassName & ); // 前缀
< Type > operator ++(ClassName &,int); // 后缀
```

在实现中，后缀操作会先构造一个临时对象，并将原对象保存，然后完成自增操作，最后将保存对象原值的临时对象返回。代码如下所示：

```
ClassName & ClassName::operator++()
{
  ClassAdd (1); //increment  current object
  return *this; //return by reference the current object
}

ClassName ClassName::operator++(int unused)
{
  ClassName temp(*this); //copy of the current object
  ClassAdd (1); //increment current object
  return temp;  //return copy
}
```

由于前缀操作省去了临时对象的构造，因此它在效率上优于后缀操作。不过，在应用到整型和长整型的操作时，前缀和后缀操作在性能上的区别通常是可以忽略的。但对于用户自定义类型，这还是非常值得注意的。当然就像 80-20 规则[⊖]告诉我们的那样，如果在

⊖　80-20 规则：一个典型的程序将花去 80% 的时间仅仅运行 20% 的代码。

一个很大的程序里，程序数据结构和算法不够优秀，它所能带来的效率提升也是微不足道的，不能使大局有所改变。但是既然它们有差异，我们为什么不在必要的时候采用更有效率的呢？

请记住：

对于整型和长整型的操作，前缀操作和后缀操作的性能区别通常是可以忽略的。对于用户自定义类型，优先使用前缀操作符。因为与后缀操作符相比，前缀操作符因为无须构造临时对象而更具性能优势。

建议 13：掌握变量定义的位置与时机

在 C/C++ 代码中，变量是一个不得不提的关键词。变量在程序中起着不同寻常的作用。所有的代码中肯定离不开各种类型变量的影子，既有内置类型的，又有自定义类型的。虽然常见，但使用它也是有一定的技巧与玄机的。掌握了这些技巧与玄机，在合适的时机将变量定义在合适的位置上，会使代码变得更具可读性与高效性。

C++ 规则允许在函数的任何位置定义变量，当程序执行到变量定义的位置，并接收到这一变量的定义时，就会调用相应的构造函数，完成变量的构造。当程序控制点超出变量的作用域时，析构函数就会被调用，完成对该变量的清理。而对象的构造和析构不可避免地会带来一定的开销，无论该变量在程序中有没有发挥作用，所以建议在需要使用变量时再去定义。

分析下面代码片段中定义的函数：

```
std::string ChangToUpper(const std::string& str)
{
  using namespace std;
  string upperStr;
  if (str.length() <= 0 )
  {
    throw error("String to be changed is null");
  }
  ...    // 将字符变为大写
  return upperStr;
}
```

在上面的代码中，变量 upperStr 定义的时机有点早。如果输入字符串为空，函数抛出异常，这个变量就不会被使用。所以，如果函数抛出了异常，就要为 upperStr 的构造与析构付出代价，而这些代价完全完全是可以避免的。所以，变得精明些，把握变量定义的时机：尽量晚地去定义变量，直到不得不定义时。代码如下所示。

```
std::string ChangToUpper(const std::string& str)
{
  using namespace std;
  if (str.length() <= 0 )
  {
    throw error("String to be changed is null");
  }
  string upperStr;
  ...    // 将字符变为大写
  return upperStr;
}
```

关于变量定义的位置，建议变量定义得越“local”越好，尽量避免变量作用域的膨胀。这样做不仅可以有效地减少变量名污染，还有利于代码阅读者尽快找到变量定义，获悉变量类型与初始值，使阅读代码更容易。

针对“变量名污染”，最臭名昭著的例子就是在 VC 6.0 环境的 for 语句中声明变量 i：

```
for( int i=0; i<N; i++)
{
   ...// do something
}
... // some code
for( int i=0; i<M; i++)
{
   ...// do another thing
}
```

上述代码在 VC 6.0 中是不能通过编译的，编译器会提示变量 i 重复定义。不熟悉 VC 6.0 环境的人肯定会很诧异。这是因为在 VC 6.0 中，i 的作用域超出了本身的循环。幸好，微软意识到了这个问题，在其后续的 VC++ 系列产品中，i 的作用域重新被限定在了 for 循环体中。

不过在这条规则中，还有一个小小的例外，如下所示：

```
for (int i = 0; i < 1000000; ++i)
{
   ClassName obj;
   obj.DoSomething();
}
```

以上变量的定义遵循了“尽可能晚，尽可能 local”的规则，但是 ClassName 的构造和析构却因此被调用了 1 000 000 次。更高效的方式就是将 obj 的定义放在循环之外，构造函数和析构函数的调用次数则会减少到 1 次：

```
ClassName obj;
for (int i = 0; i < 1000000; ++i)
{
```

```
    obj.DoSomething();
}
```

请记住：

在定义变量时，要三思而后行，掌握变量定义的时机与位置，在合适的时机于合适的位置上定义变量。尽可能推迟变量的定义，直到不得不需要该变量为止；同时，为了减少变量名污染，提高程序可读性，尽量缩小变量的作用域。

建议 14：小心 typedef 使用中的陷阱

typedef 本来是很好理解的一个概念，但是因为与宏并存，理解起来就有点困难了。再加上部分教材以偏概全，更是助长了错误认识的产生。某些教材介绍 typedef 时会给出类似以下的形式，但是缺少进一步的解释：

```
typedef string NAME;
typedef int AGE;
```

这种形式让我不由地想起 C 语言中著名的宏定义：

```
#define MAC_NAME string
#define MAC_AGE   int
```

因为二者的声明方式太相似了，所以很多人习惯用 #define 的思维方式来看待 typedef，认为应当把 int 与 AGE 看成独立的两部分。实际情况是怎样的呢？首先分析下面的代码片段：

```
typedef int* PTR_INT1;
#define int* PTR_INT2
int main()
{
    PTR_INT1 pNum1, pNum2;
    PTR_INT2 pNum3, pNum4;
    int year = 2011;
    pNum1 = &year;
    pNum2 = &year;
    pNum3 = &year;
    pNum4 = &year;
    cout<<pNum1<<" "<<pNum2<<" "<<pNum3<<" "<<pNum4;
    return 0;
}
```

输出为：2011 2011 2011 0E8951241。通过程序执行结果可以看出 typedef 与 #define 的不同：typedef 后面是一个整体声明，是不能分割的部分，就像整型变量声明 int i;，只不过 typedef 声明的是一个别名。宏定义只是简单的字符串替换，不过，typedef 并不是原地扩展，它的新名称具有一定的封装性，更易于定义变量，它可以同时声明指针类型的多个对象，而

宏则不能。使用 typedef 声明多个指针对象，形式直观，方便省事：

```
char *pa, *pb, *pc, *pd;  // 方式 1

typedef char* PTR_CHAR;
PTR_CHAR pa, pb, pc, pd;  // 方式 2，直观省事
```

除此之外，typedef 还有多种用途，下面来看看。

（1）在部分较老的 C 代码中，声明 struct 对象时，必须带上 struct 关键字，即采用“struct 结构体类型 结构体对象”的声明格式。例如：

```
struct tagRect
{
    int width;
    int length;
};
strcut tagRect rect;
```

为了在结构体使用过程中，少写声明头部的 struct，于是就有人使用了 typedef：

```
typedef struct tagRect
{
     int width;
     int length;
}RECT;
RECT rect;
```

在现在的 C++ 代码中，这种方式已经不常见，因为对于结构体对象的声明已经不需要使用 struct 了，可以采用“结构体类型 结构体对象”的形式。

（2）用 typedef 定义一些与平台无关的类型。例如在标准库中广泛使用的 size_t 的定义：

```
#ifndef _SIZE_T_DEFINED
#ifdef  _WIN64
typedef unsigned __int64    size_t;
#else
typedef _W64 unsigned int   size_t;
#endif
#define _SIZE_T_DEFINED
#endif
```

（3）为复杂的声明定义一个简单的别名。这一点将在建议 93 中详细介绍。它可以增强程序的可读性和标识符的灵活性，这也是它最突出的作用。

在 typedef 的使用过程中，还必须记住：typedef 在语法上是一个存储类的关键字，类似于 auto、extern、mutable、static、register 等，虽然它并不会真正影响对象的存储特性，如：

```
typedef static int INT2; // 不可行，编译将失败
```

编译器会提示“指定了一个以上的存储类型”。

请记住：

区分 typedef 与 #define 之间的不同；不要用理解宏的思维方式对待 typedef，typedef 声明的新名称具有一定的封装性，更易定义变量。同时还要注意它是一个无“现实意义”的存储类关键字。

建议 15：尽量不要使用可变参数

在某些情况下我们希望函数参数的个数可以根据实际需要来确定，所以 C 语言中就提供了一种长度不确定的参数，形如：“...”，C++ 语言也继承了这一语言特性。在采用 ANSI 标准形式时，参数个数可变的函数的原型是：

```
type funcname(type para1, type para2, ...);
```

这种形式至少需要一个普通的形式参数，后面的省略号（...）不能省去，它是函数原型必不可少的一部分。典型的例子有大家熟悉的 printf()、scanf() 函数，如下所示的就是 printf() 的原型：

```
int printf( const char *format ,  ... );
```

除了参数 format 固定以外，其他参数的个数和类型是不确定的。在实际调用时可以有以下形式：

```
int year = 2011;
char str[] = "Hello 2011";
printf("This year is %d", year);
printf("The greeting words are %s", str);
printf("This year is %d ,and the greeting words are:%s", year, str);
```

也许这些已经为大家所熟知，但是可变参数的实现原理却是 C 语言中比较难理解的一部分。在标准 C 语言中定义了一个头文件，专门用来对付可变参数列表，其中，包含了一个 va_list 的 typedef 声明和一组宏定义 va_start、va_arg、va_end，如下所示：

```
// File: VC++2010 中的 stdarg.h
#include <vadefs.h>

#define va_start _crt_va_start
#define va_arg _crt_va_arg
#define va_end _crt_va_end


// File: VC++2010 中的 vadefs.h
#ifndef _VA_LIST_DEFINED
```

```
typedef char *  va_list;
#define _VA_LIST_DEFINED
#endif

#ifdef  __cplusplus
#define _ADDRESSOF(v)   ( &reinterpret_cast<const char &>(v) )
#else
#define _ADDRESSOF(v)   ( &(v) )
#endif

#ifdefined(_M_IX86)
#define _INTSIZEOF(n)   ( (sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1) )
#define _crt_va_start(ap,v)  ( ap = (va_list)_ADDRESSOF(v) + _INTSIZEOF(v) )
#define _crt_va_arg(ap,t)    ( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )
#define _crt_va_end(ap)      ( ap = (va_list)0 )
```

定义 _INTSIZEOF(n) 是为了使系统内存对齐；va_start(ap, v) 使 ap 指向第一个可变参数在堆栈中的地址，va_arg(ap,t) 使 ap 指向下一个可变参数的堆栈地址，并用 * 取得该地址的内容；最后变参获取完毕，通过 va_end(ap) 让 ap 不再指向堆栈，如图 1-3 所示。

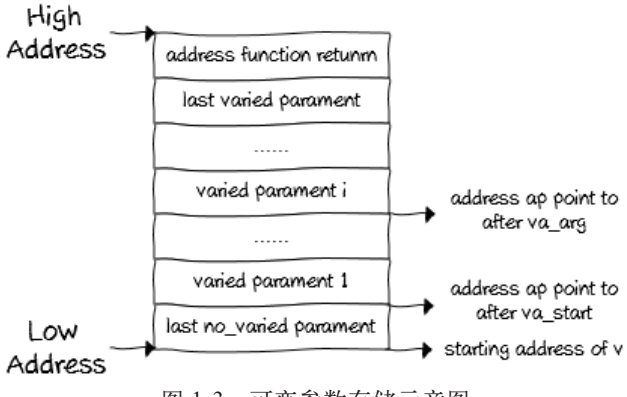

图 1-3　可变参数存储示意图

由于将 va_start、va_arg、va_end 定义成了宏，可变参数的类型和个数在该函数中完全由程序代码控制，并不能智能地进行识别，所以导致编译器对可变参数的函数原型检查不够严格，难于查错，不利于写出高质量的代码。

参数个数可变具有很多的优点，为程序员带来了很多的方便，但是上面 C 风格的可变参数却存在着如下的缺点：

（1）缺乏类型检查，类型安全性无从谈起。“省略号的本质是告诉编译器‘关闭所有检查，从此由我接管，启动 reinterpret_cast’”，强制将某个类型对象的内存表示重新解释成另外一种对象类型，这是违反“类型安全性”的，是大忌。

例如，自定义的打印函数。

```
void UserDefinedPrintFun(char* format, int i, ...)
{
    va_list arg_ptr;
    char *s = NULL;
    int *i = NULL;
    float *f = NULL;

    va_start(arg_ptr, i);
    while(*format!='\0')
    {
        format++;
        if(*(format-1)=='%' && *format=='s')
          {
            s = va_arg(arg_ptr, char*);
            …… // 输出至屏幕
          }
          else if(*(format-1)=='%' && *format=='d')
          {
            i = va_arg(arg_ptr, int*);
            …… // 输出至屏幕
          }
          else if(*(format-1)=='%' && *format=='f')
          {
            f = va_arg(arg_ptr, float*);
            …… // 输出至屏幕
          }
   }

   va_end(arg_ptr);
   return;
}
```

如果采用下面三种方法调用，合法合理：

```
UserDefinedPrintFun ("%d", 2010);  // 结果 2010
UserDefinedPrintFun ("%d%d", 2010,2011);  // 结果 20102011
UserDefinedPrintFun ("%s%d", "Hello", 2012); // 结果 Hello2012
```

但是，当给定的格式字符串与参数类型不对应时，强制转型这个“怪兽”就会被唤醒，悄悄地毁坏程序的安全性，这可不是什么高质量的程序，如下所示：

```
UserDefinedPrintFun ("%d", 2010.80f);
// 结果 2010
UserDefinedPrintFun ("%d%d", "Hello", 2012);
// 结果 150958722015（这是什么结果？？？）
```

（2）因为禁用了语言类型检查功能，所以在调用时必须通过其他方式告诉函数所传递参数的类型，以及参数个数，就像很多人熟知的 printf() 函数中的格式字符串 char* format。这种方式需要手动协调，既易出错，又不安全，上面的代码片段已经充分说明了

这一点。

（3）不支持自定义数据类型。自定义数据类型在 C++ 中占有较重的地位，但是长参数只能传递基本的内置类型。还是以 printf() 为例，如果要打印出一个 Student 类型对象的内容，对于这样的自定义类型，该用什么格式的字符串去传递参数类型呢？如下所示：

```
class Student
{
public:
   Student();
   ~ Student();
private:
   string m_name;
   char   m_age;
   int    m_scoer;
};

Student XiaoLi;
printf(format, XiaoLi);  // format 应该是什么呢
```

上述缺点足以让我们有了拒绝使用 C 风格可变参数的念头，何况 C++ 的多态性已经为我们提供了实现可变参数的安全可靠的有效途径呢！如下所示：

```
class PrintFunction
{
public:
     void UserDefinedPrintFun(int i);
     void UserDefinedPrintFun(float f);
     void UserDefinedPrintFun(int i, char* s);
     void UserDefinedPrintFun(float f, char* s);
private:
      ......
};
```

虽然上述设计不能像 printf() 函数那样灵活地满足各种各样的需求，但是可以根据需求适度扩充函数定义，这样不仅能满足需求，其安全性也是毋庸置疑的。舍安全而求危险，这可不是明白人所为。如果还对 printf() 的灵活性念念不忘，我告诉大家，有些 C++ 库已经使用 C++ 高级特性将类型安全、速度与使用方便很好地结合在一起了，比如 Boost 中的 format 库，大家可以尝试使用。

请记住：

编译器对可变参数函数的原型检查不够严格，所以容易引起问题，难于查错，不利于写出高质量的代码。所以应当尽量避免使用 C 语言方式的可变参数设计，而用 C++ 中更为安全的方式来完美代替之。

建议 16：慎用 goto

如果说有一个关键字在 C/C++ 语言程序中备受争议，那么非程序跳转关键字 goto 莫属。在早期的 BASIC 和 FORTRAN 语言中，goto 备受依赖，为了照顾部分程序员的设计习惯，在 C 语言中 goto 关键字依然得到了保留。然而，与前面两种语言有所不同，goto 在 C/C++ 语言中就像一个多余的外来户，有没有它几乎不影响 C 语言程序的设计与运行，它没有带来太大的正面作用，相反却容易破坏程序的结构性。所以，Kernighan 和 Ritchie 认为 goto 语句“非常容易被滥用”，并且建议“一定要谨慎使用，或者干脆不用”。

之所以建议避免使用 goto，是因为 C/C++ 语言中提供了更好的方式去实现 goto 的功能。为了帮助部分程序员克服 goto 依赖症，接下来我会分别介绍以下情形下 goto 语句的替换方式：

❑ if 控制内的多条语句

如果熟悉旧风格的 BASIC 和 FORTRAN，会知道只有紧跟在 if 条件后的那一条语句才属于该 if 的控制域，所以就出现了以下 goto 的使用形式：

```
Size = 20;
Flag = 1;
if( price > 15)
    goto A_PLAN;
goto B_PLAN;
A_PLAN:
    Size /=2;
    Flag = 3;
B_PLAN:
    Money = price * Size * Flag;
```

而在 C/C++ 语言中，复合语句或代码块很容易实现上述目的，而且使代码更加清晰可读，如下所示：

```
Size = 20;
Flag = 1;
if( price > 15)
{
   Size /=2;
   Flag = 3;
}
Money = price * Size * Flag;
```

❑ 不确定循环

首先请看如下代码：

```
ReadScore:
```

```
scanf( " %d " , &Score);
if(Score < 0 )
    goto ErrorStage;
... // Processing codes
goto ReadScore;
ErrorStage:
... // Error Processing
```

这种情形可以用我们熟知的 while 循环来完美代替：

```
scanf( " %d " , &Score);
while(Score >= 0)
{
    ... // Processing codes
    scanf( " %d " , &Score);
}
```

此外，如果跳转到循环结尾会开始新一轮循环，可以使用 continue 代替；如果要跳出循环，那就使用 break。

上述 goto 语句破坏了程序的结构性，影响了程序的可读性，这在 C/C++ 程序员看来是难以容忍的。然而，有一种 goto 的使用情形为许多 C/C++ 程序员所接受，那就是程序在一组嵌套循环中出现错误，无路可走时的跳转处理，如下所示：

```
while(... )
{
    for(...)
    {
           for(...)
           {
                  Processing statement;
                  if(error)
                         goto ERROR;
           }
           More processing statement;
   }
   Yet more processing statement;
}
And more processing statement;
ERROR:
    Deal_With_Error Statement;
```

请记住：

过度使用 goto 会使代码流程错综复杂，难以理清头绪。所以，如果不熟悉 goto，不要使用它；如果已经习惯使用它，试着不去使用。

建议 17：提防隐式转换带来的麻烦

在 C/C++ 语言的表达式中，允许在不同类型的数据之间进行某一操作或混合运算。当对不同类型的数据进行操作时，首先要做的就是将数据转换成为相同的数据类型。C/C++ 语言中的类型转换可以分为两种，一种为隐式转换，而另一种则为建议 11 中提及的显式强制转型。显式强制转型在某种程度上还有一定的优点，对于编写代码的人来说使用它能够很容易地获得所需类型的数据，对于阅读代码的人来说可以从代码中获知作者的意图。而隐式转换则不然，它让发生的一切变得悄无声息，在编译时这一切由编译程序按照一定规则自动完成，不需任何的人为干预。

存在大量的隐式转换也是 C/C++ 常受人诟病的焦点之一。隐式转换虽然带来了一定的便利，使编码更加简洁，减少了冗余，但是这些并不足以让我们完全接受它，因为隐式转换所带来的副作用不可小觑，它通常会使我们在调试程序时毫无头绪。就像下面的代码片段所示：

```
void Function(char c);

int main()
{
    long para = 256;
    Function(para);
    return 0;
}
```

上述代码片段中的函数调用不会出现任何错误，编译器给出的仅仅是一个警告。可是细心的程序员一眼就能看出问题：函数 Function（char c）的参数 c 是一个 char 型，256 绝不会出现在其取值区间内。但是编译器会自动地完成数据截断处理。编译器悄悄完成的这种转换存在着很大的不确定性：一方面它可能是合理的，因为尽管类型 long 大于 char，但 para 中很可能存放着 char 类型范围内的数值；另一方面 para 的值的确可能是 char 无法容纳的数据，这种“暗地里的勾当”一不小心便会造成一个非常隐蔽、难以捉摸的错误。

C/C++ 隐式转换主要发生在以下几种情形。

❑ 基本类型之间的隐式转换

```
int ival = 3;
double dval = 3.1415
cout<<(ival + dval)<<endl; //ival 被提升为 double 类型 :3.0

extern double sqrt(double);
sqrt(2); //2 被提升为 double 类型 : 2.0
```

在编译这段代码时，编译器会按照规则自动地将 ival 转换为与 dval 相同的 double 类型。

C 语言规定的转换规则是由低级向高级转换。两个通用的转换原则是：

（1）为防止精度损失，类型总是被提升为较宽的类型。

（2）所有含有小于整型类型的算术表达式在计算之前其类型都会被转换成整型。

这两点在 C++ 中依旧有效，这已无须多言。它最直接的害处就是有可能导致重载函数产生二义性，如下所示：

```
void Print(int ival);
void Print(float fval);

int ival = 2;
float fval = 2.0f;
Print(ival); // OK, int-version
Print(fval);  // OK, float-version
Print(1); // OK, int-version
Print(0.5); // ERROR!!
```

参数 0.5 应该转换为 ival 还是 fval ？这是编译器没法搞明白的一个问题。

❑ T* 指针到 void* 的隐式转换

在 C 语言中，标准允许 T* 与 void* 之间的双向转换，这也就间接导致了各种数据类型之间的隐式转换是被允许的，无论是从低级到高级，还是从高级到低级。这样的转换存在着太多的不安全因素，所以到了 C++ 中，双向变单向，只允许 T* 隐式地转换为 void* 了，示例代码如下所示：

```
char* pChar = new char[20];
void * pVoid = pChar;
```

❑ non-explicit constructor 接受一个参数的用户定义类对象之间隐式转换先看如下代码：

```
class A
{
public:
    A(int x):m_data(x){}
private:
    int m_data;
}
void DoSomething(A aObject);
DoSomething(20);
```

在上面的代码中，调用 DoSomething() 函数时会发现实参与形参类型不一致，但是因为类 A 的构造函数只含有一个 int 类型的参数，所以编译器会以 20 为参数调用 A 的构造函数，以便构造临时对象，然后传给 DoSomething() 函数。不要为此而感到惊讶，其实编译器比想像的还要聪明：当无法完成直接隐式转换的时候，它不会罢休，它会尝试使用间接的方式。所以，下面的代码也是可以被编译器接受的：

```
void DoSomething(A aObject);
```

```
float fval = 20.0f;
DoSomething(fval);
```

这是一个多么奇妙的世界。这样的隐式转换在某些时候会变得相当微妙，一个误用也许会引起难以捉摸的错误。另外，由于在隐式转换过程中需要调用类的构造函数、析构函数，如果这种转换代价很大，那么这样的隐式转换将会影响系统性能。

当然，我们熟知的隐式转换还包括“子类到基类的隐式转换”和“const 到 non-const 的同类型隐式转换”。不过这两种转换是比较安全的，所以在这里就不再详细讨论。

如果试图禁止所有的隐式类型转换，那么为了维持函数使用代码的简洁性，函数必须对所有的类型执行重载。这将是一个十分庞大且毫无技术含量的重复性工程，这不仅大大增加了函数实现的负担，重复的代码也严重偏离了 DRY 原则。

说明 DRY—— Don’t Repeat Yourself Principle，直译为“不要重复自己”。简而言之，就是不要写重复的代码。DRY 利用的方法就是抽象：把共同的事物抽象出来，把代码抽取到一个地方去，这样就可以避免重复写代码。

C/C++ 对于这个问题采取的策略是“把问题交给程序员全权处理”。程序员既然享受了隐式变换所带来的便利，那么如果出现错误也是程序员需要负责的问题。权利与义务对等，这也算得上合情合理。但在程序员的眼里，这样的处理方式却不能让他们满意。后来 C++ 设计者意识到了这个问题，于是提供了控制隐式转换的两条有效途径：

❑ 使用具名转换函数

来看一段代码：

```
class Rational
{
public:
   Rational(int numerator = 0, int denominator = 1)
         :m_num(numerator),m_den(denominator){}

   operator double() const
   {
    return ((double)m_num/(double)m_den);
   }
private:
    int m_num;
    int m_den;
};

Rational r(1,2);
cout<<r<<endl;
```

上面代码的本意是打印类似 n/m 的形式，可是结果输出的却是 0.5。问题出现在哪里？

当调用 operator<< 时，编译器会发现没有合适的函数存在，所以它就试图找到一个合适的隐式类型转换顺序，以使函数得到正常调用。本来程序中并不存在将 Rational 转为其他类型的转换规则，但是 Rational::operator double 函数告诉编译器 Rational 类型可以转换为 double 类型，所以就有了上述结果的出现。为了避免此类问题的出现，建议使用非 C/C++ 关键字的具名函数，代码如下所示：

```
class Rational
{
public:
    Rational(int numerator = 0, int denominator = 1);
    operator as_double() const;
private:
    int m_num;
    int m_den;
};

Rational r(1,2);
cout<<r<<endl; // 提示无 operator<<Rational 重载函数
```

❑ 使用 explicit 限制的构造函数

这种方式针对的是具有一个单参数构造函数的用户自定义类型。代码如下所示：

```
class Widget
{
public:
    Widget( unsigned int factor);
    Widget( const char* name, const Widget* other = NULL);
};
```

上述代码片段中，用户自定义类型 Widget 的构造函数可以是一个参数，也可以是两个参数。具有一个参数时，其参数类型可以是 unsigned int，亦可以是 char*。所以这两种类型的数据均可以隐式地转换为 Widget 类型。控制这种隐式转换的方法很简单：为单参数的构造函数加上 explicit 关键字：

```
class Widget
{
   explicit Widget(unsigned int factor);
   explicit Widget(const char* name, const Widget* other = NULL);
};
```

请记住：

提防隐式转换所带来的微妙问题，尽量控制隐式转换的发生；通常采用的方式包括：（1）使用非 C/C++ 关键字的具名函数，用 operator as_T() 替换 operato T()（T 为 C++ 数据类型）。（2）为单参数的构造函数加上 explicit 关键字。

建议 18：正确区分 void 与 void*

void 及 void 指针类型对于许多 C/C++ 语言初学者，甚至是部分有经验的程序员来说都是一个谜，它让人云里雾里，不甚清晰，因此在使用时也会出现一些这样那样的问题。也许在进入 C/C++ 语言精彩世界的第一刻就认识了 void 和 void*，可是它们的具体含义到底是什么呢？

void 是“无类型”，所以它不是一种数据类型；void * 则为“无类型指针”，即它是指向无类型数据的指针，也就是说它可以指向任何类型的数据。

从来没有人会定义一个 void 变量，如果真的这么做了，编译器会在编译阶段清晰地提示，“ illegal use of type 'void'”。void 体现的是“有与无”的问题，要先“有”了，在非 void 的前提下才能去讨论这个变量是什么类型的，此哲学思想渗透于小小 void 的使用与设计中。

void 发挥的真正作用是限制程序的参数与函数返回值。在 C/C++ 语言中，对 void 关键字的使用做了如下规定：

（1）如果函数没有返回值，那么应将其声明为 void 类型。

在 C 语言中，凡不加返回值类型限定的函数，就会被编译器作为返回整型值处理。但是许多程序员却误以为其为 void 类型。例如：

```
Add ( int a, int b );
int main()
{
    printf ( "1010 + 1001 = %d", Add ( 1010, 1001) );
    return 0;
}
Add ( int a, int b )
{
    return a + b;
}
```

程序运行的结果为：2 + 3 = 5。这个结果更加明确地说明了函数返回值为 int 类型，而非 void。

在林锐博士的《高质量程序设计指南——C++/C 语言（第 3 版）》一书中曾提到：“C++ 语言有很严格的类型安全检查，不允许上述情况（指函数不加类型声明）发生”。但是在一些较老的编译器（比如 VC++6.0）中上述 Add 函数的编译无错也无警告且运行正确，所以不能将严格的类型检查这样的重任完全交给编译器。

为了避免出现混乱，在编写 C/C++ 程序时，必须对任何函数都指定其返回值类型。如果函数没有返回值，则要声明为 void。这既保证了程序良好的可读性，也满足了编程规范性的要求。

（2）如果函数无参数，那么声明函数参数为 void。

正如我们原先遇到的情况一样，如果在调用一个无参数函数时，一不小心为其设定了参数：

```
int TestFunction(void)
{
    return 2012;
}
int main()
{
    int thisYear = TestFunction(2011);
    // processing code
    return 0;
}
```

那么在 C++ 编译器中编译代码时则会出错，提示“' TestFunction ' : function does not take 1 parameters”。而在 C 语言中，据说它能编译通过且能正确执行。之所以说是据说，是因为本人没有在 C 环境下实验证实这种情况，请原谅我的懒惰，因为我真的不想去碰 Turbo C，虽然那也曾经是我的入门开发环境。

所以，在 C/C++ 中，若函数不接受任何参数，一定要指明参数为 void。就算写的是 C 函数，为了将来的兼容性，请不要省略这个 void。

接下来说说特殊指针类型 void*。

众所周知，如果存在两个类型相同的指针 pInt1 和 pInt2，那么我们可以直接在二者间互相赋值；如果是两个指向不同数据类型的指针 pInt 和 pFloat，直接相互赋值则会编译出错，必须使用强制转型运算符把赋值运算符右侧的指针类型转换为左侧的指针类型，这一点在建议 11 中已经解释得很清晰，代码如下所示：

```
int *pInt;
float *pFloat;
pInt = pFloat;  // 编译出错，提示“'=' : cannot convert from 'int *' to 'float *'”
pInt = (float *)pFloat; // 正确，需强制转型
```

而 void * 则不同，任何类型的指针都可以直接赋值给它，无须强制转型，如下所示：

```
void *pVoid;
float *pFloat;
pVoid = pFloat;  // 正确，无需强制转型
```

但这种转换在 C++ 中并不是双向的，在不使用强制转型的前提下，不允许将 void * 赋给其他类型的指针，如下所示：

```
void *pVoid;
float *pFloat;
pFloat = pVoid;  // 错误，编译失败，提示“'=' : cannot convert from 'void *' to ' float *'”
```

对于一般数据类型的指针，我们可以进行加减等算法操作，但是按照 ANSI 标准，对 void 指针进行算法操作是不合法的：

```
// 分别采用 VC++ 编译器和 Gcc 编译器进行验证
```

```
int * pInt;
pInt ++;            // 正确，pInt 指针增大 sizeof(int)
pInt += 2;          // 正确 , pInt 指针增大 2*sizeof(int)

void * pVoid;
pVoid ++;           // 错误 ,error C2036: "pVoid*": 未知的大小
pVoid += 1;         // 错误
```

ANSI 标准之所以这样认定，是因为只有在确定了指针指向数据类型的大小之后，才能进行算法操作。但是大名鼎鼎的 GNU 则有不同的规定，它指定 void * 的算法操作与 char * 一致。所以在上面代码片段中出现错误的代码在 GNU 编译器中能顺利通过编译，并且能正确执行。虽然 GNU 较 ANSI 更开放，提供了对更多语法的支持，但是 ANSI 标准更加通用，更加“标准”，所以在实际设计中，还是应该尽可能地迎合 ANSI 标准。在实际的程序设计中，为迎合 ANSI 标准，并提高程序的可移植性，可以采用以下方式进行代码设计：

```
void * pVoid;
(char *)pVoid ++;          // ANSI：正确；GNU：正确
(char *)pVoid += 2;        // ANSI：错误；GNU：正确
```

如果函数的参数可以是任意类型指针，那么应声明其参数为 void *，最典型的例子就是我们熟知的内存操作函数 memcpy 和 memset 的原型：

```
void * memcpy(void *dest, const void *src, size_t len);
void * memset ( void * buffer, int c, size_t num );
```

仔细品味，就会发现这样的函数设计是多么富有学问，任何类型的指针都可以传入 memcpy 和 memset 中，传出的则是一块没有具体数据类型规定的内存，这也真实地体现了内存操作函数的意义。如果类型不是 void *，而是 char *，那么这样的 memcpy 和 memset 函数就会与数据类型产生明显联系，纠缠不清，这不是一个通用的、“纯粹的、脱离低级趣味”的函数设计！

请记住：

void 与 void* 是一对极易混淆的双胞胎兄弟，但是它们在骨子里却存在着质的不同，区分它们，按照一定的规则使用它们，可以提高程序的可读性、可移植性。仔细体会，还会发现隐藏在它们背后的设计哲学。

第 2 章　从 C 到 C++，需要做出一些改变

C++ 语言之父当初设计该语言的初衷是“a better C”，所以 C++ 一般被认为是 C 的超集合，但是不要因此而误以为，“这意味着 C++ 兼容 C 语言的所有东西”。作为一种欲与 C 兼容的语言，C++ 保留了一部分过程式语言的特点，大部分的 C 代码可以很轻易地在 C++ 中正确编译，但仍有少数差异，导致某些有效的 C 代码在 C++ 中无法通过编译。

因此，从 C 到 C++，我们要因为这些差异而做出一些改变，我们应当熟悉这些差异，使用原有的丰富的 C 库为现在的 C++ 工程更好地服务。

建议 19：明白在 C++ 中如何使用 C

首先，分析下面的代码片段：

```
// Demo.h
#ifndef SRC_DEMO_H
#define SRC_DEMO_H
extern "C"
{
...  // do something
}
#endif  // SRC_DEMO_H
```

显然，头文件中的编译宏“#ifndef SRC_DEMO_H、#define SRC_DEMO_H、#endif”的作用是防止该头文件被重复引用（详见建议 9）。那么，extern "C" 又有什么特殊的作用呢？暂且先留着这个疑问。

C++ 语言被称做“C with classes”、“a better C”或“C 的超集合”，但是并非兼容 C 语言的所有东西，两者之间的“大同”并不能完全抹杀其中的“小异”。最常见的差异就是，C 允许从 void 类型指针隐式转换成其他类型的指针，但 C++ 为了安全考虑明令禁止了此种行为。比如：如下代码在 C 语言中是有效的：

```
// 从 void* 隐式转换为 double*
double *pDouble = malloc(nCount * sizeof(double));
```

但要使其在 C++ 中正确运行，就需要显式地转换：

```
double *pDouble = (double *)malloc(nCount * sizeof(double));
```

除此之外，还有一些其他的可移植问题，比如 new 和 class 在 C++ 中是关键字，而在 C

中，却可以作为变量名。

若想在C++中使用大量现成的C程序库，就必须把它放到extern "C" { /* code */ }中。到这里，也许大家会茅塞顿开，明白本建议开始列出的代码片段中那些宏的真实作用了。当然，具有强烈好奇心的读者也许会有了新的问题：为什么加上extern "C" { /* code */ }就好使了呢？这是一个问题。下面就分析一下隐藏在这个现象背后的真实原因：C与C++具有不同的编译和链接方式。C编译器编译函数时不带函数的类型信息，只包含函数符号名字；而C++编译器为了实现函数重载，在编译时会带上函数的类型信息。假设某个函数的原型为：

```
int Function(int a, float b);
```

C编译器把该函数编译成类似_Function的符号（这种符号一般被称为mangled name），C链接器只要找到了这个符号，就可以连接成功，实现调用。C编译链接器不会对它的参数类型信息加以验证，只是假设这些信息是正确的，这正是C编译链接器的缺点所在。而在强调安全的C++中，编译器会检查参数类型信息，上述函数原型会被编译成_Function_int_float这样的符号（也正是这种机制为函数重载的实现提供了必要的支持）。在连接过程中，链接器会在由函数原型所在模块生成的目标文件中寻找_Function_int_float这样的符号。

解决上述矛盾就成了设置extern "C"这一语法最直接的原因与动力。extern "C"的作用就是告诉C++链接器寻找调用函数的符号时，采用C的方式，让编译器寻找_Function而不是_Function_int_float。

要实现在C++中调用C的代码，具体方式有以下几种：

（1）修改C代码的头文件，当其中含有C++代码时，在声明中加入extern "C"。代码如下所示：

```
/*C语言头文件：CDemo.h */
#ifndef C_SRC_DEMO_H
#define C_SRC_DEMO_H
extern "C" int Function(int x,int y);
#endif  // C_SRC_DEMO_H

/* C语言实现文件：CDemo.c */
#include " CDemo.h"
int Function ( int x, int y )
{
   ... // processing code
}

// C++调用文件
#include " CDemo.h"
int main()
```

```
{
    Function (2,3);
    return 0;
}
```

（2）在 C++ 代码中重新声明一下 C 函数，在重新声明时添加上 extern "C"。代码如下所示：

```
/*C 语言头文件：CDemo.h */
#ifndef C_SRC_DEMO_H
#define C_SRC_DEMO_H
extern int Function(int x,int y);
#endif  // C_SRC_DEMO_H

/* C 语言实现文件：CDemo.c */
#include "CDemo.h"
int Function ( int x, int y )
{
   ... // processing code
}

// C++ 调用文件
#include "CDemo.h"
extern "C" int Function(int x,int y);

int main()
{
    Function (2,3);
    return 0;
}
```

（3）在包含 C 头文件时，添上 extern "C"。代码如下所示：

```
/*C 语言头文件：CDemo.h */
#ifndef C_SRC_DEMO_H
#define C_SRC_DEMO_H
extern int Function(int x,int y);
#endif  // C_SRC_DEMO_H

/* C 语言实现文件：CDemo.c */
#include "CDemo.h"
int Function ( int x, int y )
{
   ... // processing code
}

// C++ 调用文件
extern "C" {
#include "CDemo.h"
}
```

```
int main()
{
    Function (2,3);
    return 0;
}
```

使用中，谨记：extern "C" 一定要加在 C++ 的代码文件中才能起作用。

请记住：

若想在 C++ 中使用大量现成的 C 程序库，实现 C++ 与 C 的混合编程，那你必须了解 extern "C" 是怎么回事儿，明白 extern "C" 的使用方式。

建议 20：使用 memcpy() 系列函数时要足够小心

memcpy()、memset()、memcmp() 等这些内存操作函数经常会帮我们完成一些数据复制、赋值等操作。因为在 C 语言中，无论是内置类型，还是自定义的结构类型（struct），其内存模型对于我们来说都是可知的、透明的。所以，我们可以对该对象的底层字节序列一一进行操作，简单而有效。代码片段如下所示：

```
struct STUDENT
{
    char _name[32];
    int _age;
    bool _gender;
};

STUDENT a = {"Li Lei", 20, true};
STUDENT b = {"Han MeiMei", 19, false};

int len = sizeof(STUDENT);
STUDENT c;
memset(&c, 0, len);
memcpy(&c, &a, len);

char *data = (char*)malloc(sizeof(char)*len);
memcpy(data, &b, len);
```

在 C++ 中，我们把传统 C 风格的数据类型叫做 POD（Plain Old Data）对象，即一种古老的纯数据。在 C 的世界里根本没有 POD 这一概念，因为 C 的所有对象都是 POD。一般来说，POD 对象应该满足如下特性：其二进制内容是可以随意复制的，无论在什么地方，只要其二进制内容存在，我们就能准确无误地还原出 POD 对象。正是由于这个原因，对于任何 POD 对象，我们都可以放心大胆地使用 memset()、memcpy()、memcmp() 等函数对对象的内

存数据进行操作。

然而在 C++ 中，每个人都要十二分的注意了。因为 C++ 的对象可能并不是一个 POD，所以我们无法像在 C 中那样获得该对象直观简洁的内存模型。对于 POD 对象，我们可以通过对象的基地址和数据成员的偏移量获得数据成员的地址。但是 C++ 标准并未对非 POD 对象的内存布局做任何定义，对于不同的编译器，其对象布局是不同的。而在 C 语言中，对象布局仅仅会受到底层硬件系统差异的影响。

针对非 POD 对象，其序列化会遇到一定的障碍：由于对象的不同部分可能存在于不同的地方，因而无法直接复制，只能通过手工加入序列化操作代码来处理对象数据，很麻烦。但是针对 POD 对象，这一切将变得不再困难：从基地址开始，直接按对象的大小复制数据，或传输，或存储，随意处理。

为什么 C++ 中的对象有可能不是一个 POD 呢？这还要从 C++ 的重要特征之一——动多态说起。动多态的一个基本支撑技术就是虚函数。在使用虚函数时，类的每一次继承都会产生一个虚函数表（vtable），其中存放的是指向虚函数的指针。这些虚函数表必须存放在对象体中，也就是和对象的数据存放在一起。因而，对象数据在内存里并不是以连续的方式存放的，而是被分割成了不同的部分，甚至“身首异处”[⊖]。既然对象数据不再集中在一起，如果此时再贸然使用 memcpy()、memset() 函数，那么所带来的后果将不可预计。

请记住：

要区分哪些数据对象是 POD，哪些是非 POD。由于非 POD 对象的存在，在 C++ 中使用 memcpy() 系列函数时要保持足够的小心。

建议 21：尽量用 new/delete 代替 malloc/free

在 C 语言中，我们已经熟悉利用 malloc/free 来管理动态内存，而在 C++ 中，我们又有了新的工具：new/delete。你不禁会产生疑问——有了 malloc/free 为什么还要 new/delete 呢？使用 malloc/free 和使用 new/delete 又有什么区别呢？首先来分析一下下面的代码片段：

```
class Object
{
public:
    Object()
    {
        cout << "Hello, I was born." << endl;
    }
    ~Object()
    {
```

⊖　关于虚函数的布局请参见《Inside the C++ Object Model》。

```
    cout << "Bye, I am died." << endl;
}
  void Hello()
  {
    cout  <<  "I am Object."<<endl;
  }
};

int  main()
{
    cout  <<  " Using Malloc & Free... "<<endl;
    Object* pObjectA = (Object*)malloc(sizeof(Object));
    pObjectA->Hello();
    free pObjectA;

    cout  <<  " Using New & Delete... "<<endl;
    Object* pObjectB = new Object;
    pObjectB->Hello();
    delete pObjectB;

    return  0;
}
```

代码运行的结果为：

```
Using Malloc & Free...
I am Object.
Using New & Delete...
Hello, I was born.
I am Object.
Bye, I am died.
```


通过结果我们可以得知：new/delete 在管理内存的同时调用了构造和析构函数；而 malloc/free 仅仅实现了内存分配与释放。接下来，我们进行详细讨论。

malloc/free 是 C/C++ 语言的标准库函数，而 new/delete 是 C++ 的运算符。它们都可用于申请动态内存和释放内存。

由于 malloc/free 是库函数，所以需要对应的头文件库函数支持。对于非内置数据类型的对象，用 malloc/free 无法满足创建动态对象的要求。这是因为对象在创建的同时要自动执行构造函数，对象在消亡之前则要自动执行析构函数。由于 malloc/free 不是运算符，不受编译器的控制管辖，所以不能够把执行构造函数和析构函数的任务强加于 malloc/free 上。而 new/delete 就不同了，它们是保留字，是操作符，它们和“+”、“−”、“*”、“/”有着一样的地位。new 不仅能完成动态内存分配，还能完成初始化工作，稳妥地构造对象；delete 不仅能完成内存的释放，还能进行对象的清理。举个形象的例子：通过 new 建造出来的是一栋房子，可以直接居住；而通过 malloc 申请到的仅仅是一块地皮，要想成为房子，还需要做出另外的努力。

malloc 的语法是：

```
指针名 =（数据类型 *）malloc（长度）；//（数据类型 *）表示指针
```

new 的语法是：

```
指针名 = new 类型（参数）；// 单个对象
指针名 = new 类型［个数］；// 对象数组
```

malloc 函数返回的是 void * 类型，如果写成：ClassA* p = malloc (sizeof(ClassA));，程序则无法通过编译，会抛出这样的错误信息："不能将 void* 赋值给 ClassA * 类型变量"。所以必须通过 (ClassA *) 来进行强制转型。相较而言，new 则不存在强制转型的问题，而且书写更为简单。总结起来，malloc 与 new 之间的区别主要有以下几点：

- new 是 C++ 运算符，而 malloc 则是 C 标准库函数。
- 通过 new 创建的东西是具有类型的，而 malloc 函数返回的则是 void*，需要进行强制转型。
- new 可以自动调用对象的构造函数，而 malloc 不会。
- new 失败时会调用 new_handler 处理函数，而 malloc 失败则直接返回 NULL。

free 与 delete 之间的区别则只有以下两点：

- delete 是 C++ 运算符，free 是 C 标准库函数。
- delete 可以自动调用对象的析构函数，而 malloc 不会。

针对内置类型而言，因为没有对象的构造与析构，所以 malloc/free 除了需要强制转型之外，和 new/delete 所做的工作无异，用哪一个只是涉及个人喜好而已。

```
//declaring native type
int* i1 = new int;
delete i1;

int* i2 = (int*) malloc(sizeof(int));
free(i2);

//declaring native type array
char* c1 = new char[10];
delete[] c1;

char* c2 = (char*) malloc(sizeof(char)*10);
free(c2);
```

既然提到了 malloc/free，不能不提一下 realloc。使用 realloc 函数可以重新设置内存块的大小，而在 C++ 中没有类似于 realloc 这样的替代品。如果出现上述需求，所做的就是，释放原来的内存，再重新申请。

既然 new/delete 的功能不仅赶上而且超越了 malloc/free，那为什么 C++ 标准中没有把

malloc/free 淘汰出局呢？这是因为 C++ 要遵守“对 C 兼容”的承诺，要让一些有价值的包含 malloc/free 函数库的 C 程序在 C++ 中得到重用。所以，在 C++ 中，new/delete 和 malloc/free 一直并存着。

不过，将 malloc/free 和 new/delete 混合使用绝对不是什么好主意。Remember that, to new is C++; to malloc is C; and to mix them is sin. 如果用 free 来释放通过 new 创建的动态对象，或者用 delete 释放通过 malloc 申请的动态内存，其结果都是未定义的。换句话说，不能保证它会出现什么问题。如果程序在关键时刻就因为这个在重要客户面前出现问题，那么懊悔恐怕已经来不及了。

请记住：

（1）不要企图用 malloc/free 来完成动态对象的内存管理，应该用 new/delete。

（2）请记住：new 是 C++ 的，而 malloc 是 c 的。如果混淆了它们，那将是件蠢事。所以 new/delete 必须配对使用，malloc/free 也一样。

建议 22：灵活地使用不同风格的注释

注释，可以说是计算机程序中不可或缺的一个部分，它的存在让我们阅读程序代码、理解作者意图变得相对容易（当然，这里说的是具有良好注释的代码）。在 C/C++ 语言中，存在着两种不同的注释语法：

- 旧有的 C 风格的注释：/* describe your purposes */
- 新式的 C++ 风格的注释：// describe your purposes

既然两种注释语法都有效，选择哪一种呢？ C 风格的还是 C++ 风格的呢？

很多的 C++ 书籍推荐我们使用新式的 C++ 风格注释语法，比如受 C++ 程序员顶礼膜拜的经典书籍《Effective C++》在条款 4 中的建议就是如此。为此，Scott Meyers 还给出了一定的理由——由“内嵌注释结束符”引发的“惨案”[⊖]：

```
/* C 风格的注释 */
if(a>b)
{
  /*    int temp =a;   /* swap a and b */
        a = b;
        b = temp;
  */
}
```

⊖ 此段代码取自 Scott Meyers 的《Effective C++》，在此表示感谢。

```
// C++ 风格的注释
if(a>b)
{
        //int temp =a;   // swap a and b
        //a = b;
        //b = temp;
}
```

当程序员因为某些特殊原因而采用 C 风格的注释语法将上述代码进行注释时，由于原代码中存在原有的内嵌注释，导致注释过早地找到结束匹配符，使代码注释失效，出现编译错误。而 C++ 风格的注释则不会出现类似的麻烦。

然而，正如一个硬币有两面，任何东西都是有利有弊的。让程序员更加便利与轻松才是硬道理。使用注释亦然。还是先看下面的一段代码：

```
/***
* new.cxx - defines C++ new routine
*
*  Copyright (c) Microsoft Corporation.  All rights reserved.
*
*Purpose:
*      Defines C++ new routine.
*******************************************************/
#ifdef _SYSCRT
#include <cruntime.h>
#include <crtdbg.h>
#include <malloc.h>
#include <new.h>
#include <stdlib.h>
#include <winheap.h>
#include <rtcsup.h>
#include <internal.h>

void * operator new( size_t cb )
{
   void *res;
   for (;;) {
       //  allocate memory block
       res = _heap_alloc(cb);
       //  if successful allocation, return pointer to memory
       if (res)
            break;
       //  call installed new handler
       if (!_callnewh(cb))
            break;
       //  new handler was successful -- try to allocate again
    }
    RTCCALLBACK(_RTC_Allocate_hook, (res, cb, 0));
    return res;
```

```
}
#else  /* _SYSCRT */
```

这是 VC++ 库中 new.cpp 文件中的部分代码。在注释方面，这段代码中有很多值得我们学习的地方。

❑ 版权和版本声明，使用 C 风格的 /* */

标准化的代码有很多必不可少的东西，比如版权信息、文件名称、标识符、摘要、当前版本号、作者 / 修改者、完成日期、版本历史信息，等等。这些信息不会为我们的代码运行带来任何的改进，但是可以提高了代码的可读性，方便代码的维护。如此繁缛的信息，可能多达十几行，此时如果使用 C++ 风格的注释语法，那么就得记得在每一行的开始都写下两个“/”符。那此时何不采用更加简单便利的 /* */ 呢？

❑ 内嵌注释用 //

内嵌注释一般出现在代码主体内。此时，建议使用新式的 C++ 风格的注释语法。最直接的原因就是避免出现那些由“内嵌注释结束符”引发的“惨案”。不过，在这种情况下，出于调试原因用 /* */ 注掉一块代码，也不会出现什么问题。

❑ 宏尾端的注释用 /* */

Scott Meyers 对于注释语法的使用还提出了一个问题：一些“古董”级的、只针对 C 编译器而写的预处理器不能识别 C++ 风格的注释，所以下面的代码就不能按照预期那样正常运行，它们会把注释当成宏的一部分：

```
#define LIGHT_SPEED 3e8  // m/sec (in a vacuum)
```

虽然使用这样的“古董”预处理器的人近乎绝迹，但是保不齐会出现一个特例。所以为了保证百分之百不出错，建议在宏尾端的注释使用 C 风格的注释语法：

```
#define LIGHT_SPEED 3e8  /* m/sec (in a vacuum) */
```

除此之外，还有一个特别的使用情形：默认参数函数的定义。代码片段如下所示：

```
// 声明文件
class A
{
public:
    void Function( int para1, int para2 = 0 );
};

// 实现文件
void A::Function( int para1, int para2 /* = 0*/ )
{
    // processing code
}
```

我们一般将类的声明与实现进行分离，放置在不同的文件之中。此时如果函数存在默认

参数，它只能出现在声明中，不过，在实现中缺少默认参数的说明可能会影响我们对函数的设计或理解，所以有必要在实现中对默认参数进行一些说明。使用 C 风格的注释语法按照上述形式进行说明确实是一个值得推荐的方式。在这种情形下，C++ 风格的注释变得无能为力了。

灵活地使用两种形式的注释方式，在保证代码鲁棒性[⊖]、可读性的同时，尽量使程序员获得更多轻松与便利。

请记住：

C 风格的注释 /* */ 与 C++ 风格的注释 // 在 C++ 语言中同时存在，所以我们可以充分地利用两种注释的长处，并注意可能存在的问题，这会让我们的编码变得更加轻松、便利、高效！

建议 23：尽量使用 C++ 标准的 iostream

IO 是我们最基本的需求之一。比如当我们进入 C++ 世界时所接触的第一个程序 HelloWorld，采用 printf() 或 operator<< 都可以。所以，我们会有如下的版本：

```
//Version 1
#include < stdio.h >
int main()
{
    printf("Hello World");
    return 0;
}

//Version 2
#include < cstdio >
int main()
{
    std::printf("Hello World");
    return 0;
}

//Version 3
#include < iostream.h >
int main()
{
    cout<<"Hello World";
```


⊖ 此处的鲁棒性不同于一般意义上的鲁棒性，这里主要是指代码可在不同平台上正确执行，不因注释而导致代码出现错误。

```
    return 0;
}

//Version 4
#include < iostream >
using namespace std;
int main()
{
    cout<<"Hello World";
    return 0;
}
```

stdio.h、cstdio 两个头文件中都有 printf() 的定义，而 iostream.h 和 iostream 中也都有 operator<< 的定义。是 stdio.h 还是 cstdio？是 iostream.h 还是 iostream？是 printf() 还是 operator<<？这些都值得思考。

关于 File.h 和 File，这还得从 C++ 标准库说起。C++ 标准程序库涵盖范围相当大，包含了许多好用的功能，所以，标准库与第三方提供程序库中的类型名称和函数名称发生名称冲突的可能性大大增加。为了避免这个问题的发生，标准委员会决定让标准库中的内容都披上 std 的外衣，放在 std 名空间中。但是这么做同时又带来了一个新的兼容性问题：很多 C++ 程序代码依赖的都是没有用 std 包装的 C++“准”标准库，例如 iostream.h 等，如果将原有的 iostream.h 贸然代替掉，那肯定会引起众多程序员的抗议。

标准化委员会最后决定设计一种新的头文件名来解决这个问题。于是，他们把 C++ 头文件 File.h 中的 .h 去掉，将 File 这样没有后缀的头文件名分配给那些用 std 包装过的组件使用；而旧有的 File.h 保持不变，仅仅在标准中声明不再支持它，顺势把问题丢给了广大厂商，堵住了那些老程序员抗议的嘴。同样，对 C 的头文件也做了相同的处理，在前面加上了一个字母 c 以示区分。因为 C++ 标准还要遵守“对 C 兼容”这个契约，备受“歧视”的旧有的 C 头文件“侥幸存活”了下来。虽然标准化委员会选择抛弃那些旧有的 C++ 头文件，但是各大厂商为了各自的商业利益，却依然选择了对旧有 C++ 头文件的支持。

因此就出现了类似 stdio.h 和 cstdio、iostream.h 和 iostream 这样的双胞胎：

```
// 标准化以前 C++ 中的 C 标准库，标准 C 的头文件继续获得支持，这类文件的内容并
// 未放在 std 中
#include<stdio.h>

// 标准化后经过改造的 C 的标准库，C 的标准库对应的新式 C++ 版本，这类头文件的
// 内容也有幸穿上了 std 的外衣
#include<cstdio>

// 标准化以前的头文件，这些头文件的内容将不处于 namespace std 中
#include<iostream.h>

// 标准化以后的标准头文件，它提供了和旧有的头文件相同的功能，但它的内容都并
```

```
// 入了 namespace std 中，从而有效避免了名称污染的问题
#include<iostream>
```

其实标准化以后标准程序库的改动并不只有这些，很多标准化的组件都被模板化了。具体参见侯捷翻译的《C++ 标准程序库》。

接下来再说说 printf() 和 operator<< 的问题。首先通过上面的讲解我们可以知道 printf() 函数继承自 C 标准库，而 operator<< 是标准 C++ 所独享的。对于 printf() 函数，大家肯定很熟悉，它是可移植的、高效的，而且是灵活的；但是正如建议 15 中所说的那样，因为 printf()、scanf() 函数不具备类型安全检查，也不能扩充，所以并不完美；而 C 语言遗留的问题在 C++ 的 operator<< 中得到了很好的解决，换句话说，printf() 的缺点正是 operator<< 的长处。现在再去回顾建议 15 中提到的关于 Student 类型对象打印的问题，用 operator<< 就可以完美解决了：

```
class Student
{
public:
    Student(string& name, int age, int scoer);
    ~ Student();
private:
    string m_name;
    int    m_age;
    int    m_scoer;
friend ostream& operator<<( ostream& s, const Student& p );
};

ostream& operator<<( ostream& s, const Student& p )
{
    s<<p. m_name<<" "<<p.m_age<<" "<<p. m_scoer;
    return s;
}
// 调用 operator<<
Student XiaoLi("LiLei", 23, 97);
cout<< XiaoLi;
```

当然了，相比 C++ iostream 程序库中的类，C 中的 stream 函数也并不是一无是处的：

（1）一般认为 C stream 函数生成的可执行文件更小，有着更高的效率；Scott Meyers 在《More Effective C++》的条款 23 中的测试也很好地证明了这一点；

（2）C++ iostream 程序库中的类会涉及对象构造、析构的问题，而 C stream 函数没有这些，所以不会像前者那样因为构造函数带来不必要的麻烦。

（3）C stream 函数有着更强的可移植能力。

对于一般应用程序而言，这三条优点还不足以打动他们，抛弃 C++ iostream 程序库，转而投向 C stream 函数。

请记住：

C++ iostream 程序库中的类与 C stream 函数虽然各有优点，但是一般推荐使用前者，因为类型安全与可扩充性对于我们更有吸引力，所以，建议使用 #include< iostream >，而不是 #include< stdio.h >、#include< cstdio >、#include< iostream.h >。

建议 24：尽量采用 C++ 风格的强制转型

在建议 11 中，我们详细讲述了强制转型存在的一些问题，并建议在代码编写过程中尽量避免使用这个招人讨厌的东西。然而，正如哲学中所讲的一样：存在的即是合理的。强制转型肯定具有它存在的意义。在某些情形下我们必须求助于这个“讨厌鬼”，以帮助我们更好地完成程序设计。

比如，const 属性的去除（请不要纠结于下面示例函数的“不良”设计）：

```
class CStudent{};
const CStudent* GetCertainStudent(const std::string& name)
{
    CStudent* p = new CStudent(name);
    return p;
}

CStudent* p = GetCertainStudent("Li Lei");
```

在 VC++ 下编译，编译器会报错：

```
error C2440: "初始化": 无法从 "const CStudent *" 转换为 "CStudent *"
```

此时我们就只能求助于 const_cast 了：

```
CStudent* p = const_cast<CStudent*>(GetCertainStudent("Li Lei"));
```

这里需要提醒的是，不要随意去除变量的 const 属性，除非是经过深思熟虑后不得不这样做。

在 C/C++ 编程中，新旧两种风格的强制转型同时存在。当强制转型已成为不可避免的定局时，安全性相对高的 C++ 风格的强制转型更为可取。

首先，新风格的强制转型不再像 C 风格的强制转型那样简单粗暴，在代码中它们更容易识别，更容易找到这些类型系统破坏者的藏匿之处。

其次，新风格的强制转型针对性更强，它针对特定的目的进行了特别的设计。如果对这些特别设计的理解不是很清晰，请返回去看看建议 11。这样能让程序员更清晰地了解强制转型的目的，同时使利用编译器诊断使用错误成为可能。

请记住：

如果实在不能避免，建议采用安全性较高的 C++ 风格的强制转型形式。新风格更容易被注意，而且具有一定的针对性。

建议 25：尽量用 const、enum、inline 替换 #define

在建议 4 中，我们已经详细说明了在使用宏时应注意的一些问题。“表面似和善、背后一长串”绝对是对宏的形象表述。宏的使用具有一些优点：能减少代码量（比如简单字符替换重复的代码），在某种程度上提供可阅读性（比如 MFC 的消息映射），提高运行效率（比如没有函数调用开销）。

然而谈到宏，绝对绕不开预处理器。把 C/C++ 源码从源文件的形式变成可执行的二进制文件通常需要三个主要步骤：预处理→编译→链接。在预处理阶段，预处理器会完成宏替换。因为此过程并不在编译过程中进行，所以难以发现潜在的错误及其他代码维护问题，这会使代码变得难以分析，繁于调试。所以，宏——这个 C 语言中的“大明星”在 C++ 的世界里却变成了程序员深恶痛绝的东西。因为 #define 的内容不属于语言自身的范畴，所以 C++ 设计者为我们提供了替代宏的几大利器，建议我们尽量使用编译器管制下的 const、enum、inline 来实现 #define 的几大功能。如此看来，本建议的名称换做“尽量把工作交给编译器而非预处理器”或许更合适。

接下来分析一下 #define 的弊端，请看下面的代码片段：

```
#define PI 3.1415926
```

在预处理阶段，预处理器就完成了代码中符号 PI 的全部替换，因为这个过程发生在源代码编译以前，所以编译器根本接触不到 PI 这个符号名，这个符号名更不会被编译器列入到符号表中。如果因为在代码中使用了这个常量 PI 而引起问题，那这个错误将可能变得不易察觉，难以找到问题，出错信息只会涉及 3.1415926，对 PI 则只字未提。

如果 PI 是在某个大家并不熟悉的或出自别人之手的头文件中定义的，那么寻找数值 3.1415926 的出处就如同大海捞针，费时费力。不过这一切也并非是不可避免的，解决的办法很简单，就是“使用常量来代替宏定义”：

```
const double PI = 3.1415926;
```

作为语言层面的常量，PI 肯定会被编译器看到，并且会确保其进入符号表中，也就不会出现类似“3.1415926 有错误”这样模糊不清的错误信息了。当出现问题时，我们也会有章可循，可以通过符号名顺藤摸瓜，消灭错误。另外，使用常量可以避免目标码的多份复制，也就是说生成的目标代码会更小。这是由于预处理器会对目标代码中出现的所有宏 PI 复制出

一份 3.1415926，而使用常量时只会为其分配一块内存。

在使用普通常量时，有一种特殊情形会让我们感觉棘手，那就是常量指针。用 const 去修饰指针的方式有多种，诸如：

```
const char* bookName = "150 C++ Tips";
char* const bookName = "150 C++ Tips";
const char* const bookName = "150 C++ Tips";
```

应该使用哪一种方式确实是一个需要明确的问题。const 修饰指针的规则可以简单地描述为：如果 const 出现在 * 左边，表示所指数据为常量；如果出现在 * 右边，表示指针自身是常量。需要注意的是，在头文件中定义常量指针时，是将指针声明为 const 了，而不是指针指向的数据。所以，如果定义一个指向常量字符串的常量指针，我们选择的就是最后一种，需要用两个 const 进行修饰。然而在定义指向常量字符串的常量指针时，用两个 const 修饰并不是我们推荐的形式。我们推荐使用更加安全、更加高级的 const string 形式：

```
const string bookName("150 C++ Tips");
```

作为 C++ 中最重要的概念，class 与很多其他的关键字都产生了联系，const 也肯定不会放过纠缠这个 C++ 主角的机会，所以就有了常量数据成员。定义常量数据成员的主要目的就是为了将常量的作用域限制在一个特定的类里，为了让限制常量最多只有一份，还必须将该常量用 static 进行修饰，例如：

```
class CStudent
{
private:
    static const int NUM_LESSONS = 5; //声明常量
    int scores[NUM_LESSONS];          //使用常量
};
```

注意，上述注释中说的是“声明常量”，而非“定义常量”，并且在声明的同时，完成了“特殊形式”的初始化。之所以谓之“特殊形式”，是因为我们熟悉的一般形式的初始化是不允许放在声明里的。这种“特殊形式”的初始化在 C++ 中被称为“类内初始化”。还有一点需要明确的是，在不同的编译器中对类内初始化的支持情况也不尽相同。在 VC++ 2010 中，并不是所有的内置类型都可以实现类内初始化，它只对整数类型（比如 int、char、bool）的静态成员常量才有效。如果静态成员变量是上述类型之外的其他类型，如 double 型，那么需要将该类的初始化放到其实现文件该变量的定义处，如下所示：

```
/* VC++ 2010 */
// CMathConstants声明文件（.h）
class CMathConstants
{
private:
    static const double PI;
```

```
};
// CMathConstants 实现文件（.cpp）
const double CMathConstants::PI = 3.1415926;
```

而在 GCC 编译器中，内置的 float、double 类型的静态成员常量都可以采用类内初始化，如下所示：

```
/* Gcc 4.3 */
// CMathConstants 声明文件（.h）
class CMathConstants
{
private:
    static const double PI = 3.1415926;
};
```

当然，如果不习惯类内初始化，讨厌其破坏了静态成员常量声明、定义的统一形式，可以选择将类内初始化全部搬到类实现文件中去，这也是我们比较推荐的形式。更何况早期的编译器可能不接受在声明一个静态的类成员时为其赋初值，那又何必去惹这些不必要的麻烦呢？

另外，如果编译器不支持类内初始化，而此时类在编译期又恰恰需要定义的成员常量值，身处如此左右为难的境地，我们该采取怎样的措施？那就求助于 enum ！巧用 enum 来解决这一问题。这一技术利用了这一点：枚举类型可以冒充整数给程序使用。代码如下所示：

```
// CStudent 声明文件（.h）
class CStudent
{
private:
    enum{ NUM_LESSONS = 5 };
    int scores[NUM_LESSONS];
};
```

需要说明的一点是，类内部的静态常量是绝对不可以使用 #define 来创建的，#define 的世界中没有域的概念。这不仅意味着 #define 不能用来定义类内部的常量，同时还说明它无法为我们带来任何封装效果。

#define 的另一个普遍的用法是“函数宏”，即将宏定义得和函数一样，就像建议 4 中的：

```
#define ADD( a, b )  ((a)+(b))
#define MULTIPLE( a, b )  ((a)*(b))
```

这样的“函数宏”会起到“空间换时间”的效果，用代码的膨胀换取函数调用开销的减少。这样的宏会带来数不清的缺点，建议 4 中已经说得很清晰。如果使用宏，必须为此付出精力，而这是毫无意义的。幸运的是，C++ 中的内联函数给我们带来了福音 ：使用内联函数的模板，既可以得到宏的高效，又能保证类型安全，不必为一些鸡毛蒜皮的小问题耗费宝贵的精力。

```
template<typename T>
inline T Add(const T& a, const T& b)
{
    Return (a+b);
}

template<typename T>
inline T Multiple(const T& a, const T& b)
{
    Return (a*b);
}
```

这一模板创建了一系列的函数，方便高效，而且没有宏所带来的那些无聊问题。与此同时，由于 Add 和 Multiple 都是真实函数，它也遵循作用域和访问权的相关规则。宏在这个方面上确实是望尘莫及。

虽然建议尽量把工作交给编译器而非预处理器，而且 C++ 也为我们提供了足以完全替代 #define 的新武器，但是预处理器并未完全退出历史舞台，并没有完全被抛弃。因为 #include 在我们的 C/C++ 程序中依旧扮演着重要角色，头文件卫士 #ifdef/#ifndef 还在控制编译过程中不遗余力地给予支持。但是如果将来这些问题有了更加优秀的解决方案，那时预处理器也许就真的该退休了。

请记住：

对于简单的常量，应该尽量使用 const 对象或枚举类型数据，避免使用 #define。对于形似函数的宏，尽量使用内联函数，避免使用 #define。总之一句话，尽量将工作交给编译器，而不是预处理器。

建议 26：用引用代替指针

指针，可以通向内存世界，让我们具备了对硬件直接操作的超级能力。C++ 意识到了强大指针所带来的安全隐患，所以它适时地引入了一个新概念：引用。引用，从逻辑上理解就是“别名”，通俗地讲就是“外号”。在建立引用时，要用一个具有类型的实体去初始化这个引用，建立这个“外号”与实体之间的对应关系。

对于引用的理解与使用，主要存在两个的问题：

- 它与指针之间的区别。
- 未被充分利用。

引用并非指针。引用只是其对应实体的别名，能对引用做的唯一操作就是将其初始化，而且必须是在定义时就初始化。对引用初始化的必须是一个内存实体，否则，引用便成为了无根之草。一旦初始化结束，引用就是其对应实体的另一种叫法了。与指针不同，引用与地

址没有关联，甚至不占任何存储空间。代码如下所示：

```
int  iNum = 12;
int &rNum = iNum;
int *pNum = &rNum;     // 等同于 int *pNum = & iNum;
iNum = 2011;           // rNum 的值也为 2011
```

由于引用没有地址，因此就不存在引用的引用、指向引用的指针或引用的数组这样的定义。据说尽管 C++ 标准委员会已经在讨论，认为应在某些上下文环境里允许引用的引用。但那都是将来的事，至少现在不可以，将来的事谁又说得准呢？

因为是别名，与实体所对应，所以引用不可能带有常量性和可挥发性。所以，下面的代码在编译时会出现问题：

```
int r = 10;
int & volatile s = r;
int & const m = r;
volatile int& t = r;
const int& n = r;
```

之所以说是出现问题，而不是错误，最主要的原因是各厂商编译器对于上述语法的容忍程度不同，有的会直接抛出错误并且编译失败，有的却只给出一个警告，比如：

❑ gcc 4.3 给出错误

```
error: volatile/const 限定符不能应用到 'int&' 上
```

❑ VC++ 2010 给出警告

```
warning C4227: 使用了记时错误 : 忽略引用上的限定符
```

对于加在引用类型前面的 const 或 volatile 修饰词，它们是符合编译器规则的，或者说被编译器选择性忽略了，没有什么问题（无 error 或 warning）。而对于指针，上述使用绝对不存在任何的问题。

C 阵营中那帮“顽固派”习惯在 C++ 工程里使用指针，并且以此为傲，现在该是为引用翻身的时候了。先看一个简单的示例：

```
void SwapData1(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

void SwapData2(int* a, int* b)
{
    int temp = *a;
    *a = *b;
```

```
    *b = temp;
}

void SwapData3(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

上述代码要实现的功能极为简单，就是交换两个数据的值。SwapData1() 是不能实现所设定的功能的，主要原因是函数内交换的只是实参的副本；而 SwapData2() 和 SwapData3() 则正确地实现了作者意图，其汇编代码如下：

```
    SwapData1(a,b);
00F431EC  mov         eax,dword ptr [b]
00F431EF  push        eax
00F431F0  mov         ecx,dword ptr [a]
00F431F3  push        ecx
00F431F4  call        SwapData1 (0F4114Fh)
00F431F9  add         esp,8
    SwapData2(&a,&b);
00F431FC  lea         eax,[b]
00F431FF  push        eax
00F43200  lea         ecx,[a]
00F43203  push        ecx
00F43204  call        SwapData2 (0F411EFh)
00F43209  add         esp,8
    SwapData3(a,b);
00F4320C  lea         eax,[b]
00F4320F  push        eax
00F43210  lea         ecx,[a]
00F43213  push        ecx
00F43214  call        SwapData3 (0F4101Eh)
00F43219  add         esp,8
```

正如汇编代码中所示的那样，SwapData2() 和 SwapData3() 其实是一样的，都是对源数据进行操作。但是相较于 SwapData2() 而言，SwapData3() 的实现代码更加简洁清晰。这就是引用在传递函数参数时所具有的巨大优势。

再来看函数返回值方面，如果其返回值是引用类型，那么就意味着可以对该函数的返回值重新赋值。就像下面代码所示的数组索引函数一样：

```
template <typename T, int n>
class Array
{
    public:
        T &operator [](int i)
        {
```

```
            return a_[i];
        }
        // ...
    private:
        T a_[n];
};

Array<int, 10> iArray;
for(int i=0; i<10; i++)
        iArray[i] = i*2;
```

当然，上述代码可以使用指针重新实现，并且可以保证实现相同的功能，但是相比指针实现版，引用返回值使对数组索引函数的操作在语法上颇为自然，更容易让人接受。

也许有人认为，指针功能更强大，因为还有指向数组的指针、指向函数的指针，这里我要说的是，引用同样可以。引用在指向数组时还能够保留数组的大小信息，关于这方面的内容，在此就不多讲了。

请记住：

从编码实践角度来看，指针和引用并无太多不同。在大多情况下，指针可由索引类型完美代替，并且其实现代码更简洁清晰，更加易于理解。

第 3 章　说一说“内存管理”的那点事儿

在 C++ 的世界里，“烫”和“屯”是我们遇到得最多的两个汉字（限于 VC 用户）。可能有人不禁要问：这是为什么呢？

答案是：在 VC 中，栈空间未初始化的字符默认是 −52，补码是 0xCC。两个 0xCC，即 0xCCCC 在 GBK 编码中就是“烫”；堆空间未初始化的字符默认是 −51，两个 −51 在 GBK 编码中就是“屯”。二者都是未初始化的内存。

C++ 赋予了我们直接面对内存、操作内存的能力，但是内存管理却一直以来被认为是 C++ 语言的一大难点。因为在 C++ 语言中，缺少 GC（垃圾回收器），内存管理需要程序员手动完成，并且还要为可能的失误承担后果。

正如下面的“代码故事”：

```
#include <stdio.h>
#include <stdlib.h>

/*
    在经历过无数的 " 烫烫烫烫烫 "," 屯屯屯屯屯屯 " 之后，
    我们都知道了：内存原来是可以驾驭的 ...
*/

int main()
{
    /*
       原来内存管理是这样的，即便结果完美无缺，
       但却危机四伏 ...
    */
    const char *src="Hello Csdn!";
    char *dest=(char*) malloc(strlen(src));
    memcpy(dest,src,strlen(src)+1);
    printf("%s\n",dest);

    return 0;
}
/*
   代码结束了，故事也到此为止。但是我们要做的还很多。
   希望我们能少遇到一点烫和屯 ...
*/
```

所以，我们要说说内存管理那点事儿，争取早日练就内存管理的高深技艺。

建议 27：区分内存分配的方式

在 C/C++ 语言中，用内存管理的水平去划分高手与菜鸟已经成为一种不成文的约定：可以从中获得更好的性能、更大自由的被称作 C++ 高手，而程序经常面临着莫名其妙的崩溃，一遍遍的调试，费时又费力的则可能是菜鸟级别的。而这一切都源于那让人又爱又恨的 C++ 内存管理的灵活性。其中，多样的内存分配方式就是其灵活性的最好例证之一。

一个程序要运行，就必须先将可执行的程序加载到计算机内存里，程序加载完毕后，就可以形成一个运行空间，并按照图 3-1 所示的那样进行布局。

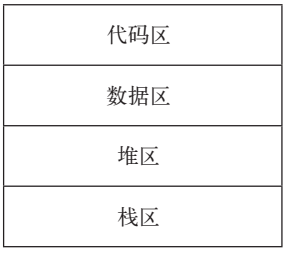

图 3-1　程序运行空间布局图

代码区（Code Area）存放的是程序的执行代码；数据区（Data Area）存放的是全局数据、常量、静态变量等；堆区（Heap Area）存放的则是动态内存，供程序随机申请使用；而栈区（Stack Area）则存放着程序中所用到的局部数据。这些数据可以动态地反应程序中对函数的调用状态，通过其轨迹也可以研究其函数机制。其中，除了代码区不是我们能在代码中直接控制的，剩余三块都是我们编码过程中可以利用的。在 C++ 中，数据区又被分成自由存储区、全局 / 静态存储区和常量存储区，再加上堆区、栈区，也就是说内存被分成了 5 个区。这 5 种不同的分区各有所长，适用于不同的情况。

❑ 栈（Stack）区

在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元将自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是所分配的内存容量有限。

❑ 堆（Heap）区

堆就是那些由 new 分配的内存块，其释放编译器不会管它，而是由我们的应用程序控制它，一般一个 new 就要对应一个 delete。如果程序员没有释放掉，那么在程序结束后，操作系统就会自动回收。

❑ 自由存储区

自由存储区是那些由 malloc 等分配的内存块，它和堆十分相似，不过它是用 free 来结束自己生命的。

❑ 全局 / 静态存储区

全局变量和静态变量被分配到同一块内存中，在以前的 C 语言中，全局变量又分为初始化的和未初始化的，在 C++ 里面没有作此区分，它们共同占用同一块内存区。

❑ 常量存储区

这是一块比较特殊的存储区，里面存放的是常量，不允许修改。

上述 5 种分区中，最常用的就是堆与栈，容易混淆的也是堆与栈。在 BBS 论坛里，

几乎到处都能看到堆与栈的争论。堆与栈的区分问题，似乎是每一个 C++ 程序员成长路上都会遇到的永恒话题。那么堆与栈之间到底有什么分别与联系呢？这就是接下来我要阐述的问题。

首先，还是分析下面的代码片段：

```
const int COUNT = 10;
void Function()
{
     string* pStr = new string[COUNT];
}
```

你是否相信就这么简单的一个函数，它却涉及了 5 种内存分区中的 3 种呢？ COUNT 是一个常量，被安置在了常量存储区，不可修改；pStr 是局部变量，理所应当地放入栈里；而通过 new string[COUNT] 获得的则是一块堆空间。多么精妙，多么不可思议！当然，上述代码片段只是一个示例，是经不起推敲的，因为它会引起内存泄露（缺少与 new 对应的 delete 去释放内存）。

似乎脱离了主题，还是言归正传，说说堆与栈的区别。总的来说，二者的区别主要有以下几个方面：

❑ 管理方式不同

对于栈来讲，它是由编译器自动管理的，无须我们手工控制；对于堆来说，它的释放工作由程序员控制，容易产生 memory leak。

❑ 空间大小不同

一般来讲在 32 位系统下，堆内存可以达到 4GB 的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定空间大小的。

❑ 碎片问题

对于堆来讲，频繁的 new/delete 势必会造成内存空间的不连续，从而产生大量的碎片，使程序效率降低。对于栈来讲，则不存在这个问题，其原因还要从栈的特殊数据结构说起。栈是一个具有严明纪律的队列，其中的数据必须遵循先进后出的规则，相互之间紧密排列，绝不会留给其他数据可插入之空隙，所以永远都不可能有一个内存块从栈中间弹出，它们必须严格按照一定的顺序一一弹出。

❑ 生长方向

对于堆来讲，其生长方向是向上的，也就是向着内存地址增加的方向增长；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长的。

❑ 分配方式

堆都是动态分配的，没有静态分配的堆。栈有两种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由 alloca 函数完成，但是栈的动态分配和堆是不同的，它的动态分配是由编译器进行释放的，无须我们手工实现。

❑ 分配效率

栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：它会分配专门的寄存器存放栈的地址，而且压栈出栈都会有专门的指令来执行，这就决定了栈的效率比较高。堆则是 C/C++ 函数库提供的，它的机制很复杂，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构 / 操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），则可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存了，然后返回。显然，堆的效率比栈要低得多。

堆和栈相比，由于堆使用了大量 new/delete，容易造成大量的内存碎片，而且它没有专门的系统支持，效率很低，另外它还可能引发用户态和核心态的切换，以及内存的申请，代价会变得很高。所以栈在程序中是应用最广泛的，就算是函数的调用也会利用栈去完成，函数调用过程中的参数、返回的地址、EBP 和局部变量都是采用栈的方式存放的。所以，我们推荐大家尽量多用栈，而不是用堆。

虽然栈有如此多的好处，但是由于和堆相比它不是那么灵活，有时候会分配大量的内存空间，在遇到这种情况时还是用堆好一些。

请记住：

内存分配具有多种不同的方式，它们各具特点，适用于不同的情形。所以，要在合适的地方采用合适的方式完成内存的分配。

建议 28：new/delete 与 new[]/delete[] 必须配对使用

operator new 和 operator delete 函数有两个重载版本：

```
void* operator new (size_t);       // allocate an object
void* operator new [] (size_t);    // allocate an array
void operator delete (void*);      // free an oject
void operator delete [] (void*);   // free an array
```

熟悉 C 语言的朋友看到这里可能会很奇怪：在 C 语言中，无论申请的是单个对象，还是一个数组，管理内存所用的都是 malloc/free，但是为什么到了 C++ 里会出现两个呢？何况建议 21 中已经说明，new/delete 在功能上比前者更加强劲。

先分析以下代码片段存在的问题：

```
class Test
{
public:
     Test() { cout << "ctor" << endl; }
```

```
    ~Test() { cout << "dtor" << endl; }
    Hello(){ cout << "Hello C++" << endl; }
};
int main()
{
    cout <<"Test 1:"<< endl;
    Test* p1 = new Test[3];
    delete p1;

    cout <<"Test 2:"<< endl;
    Test* p2 = new Test;
    delete[] p2;

    return 0;
}
```

上述代码看起来井然有序：我们采纳了建议 21，用 new 完成了内存申请，并且使用了与之对应的 delete 来释放内存。可是，执行结果却显示上述代码存在问题。在 Test 1 中，构造函数调用了 3 次，构造了 3 个 Test 类型对象，而在删除时，却只析构了一个对象 p1[0]。在 Test 2 中，构造函数调用了一次，但是析构函数却被调用了多次，将本不属于该对象的空间当成该类型对象进行了清理。也许各大厂商意识到了这个问题，于是让编译器能够检测出这样的错误。所以在 VC++2010 中，如果出现上述情形，编译器就会给出“debug assertion failed”或“堆被损坏”的错误信息。

C++ 告诉我们在回收用 new 分配的单个对象的内存空间时用 delete，在回收用 new[] 分配的一组对象的内存空间时用 delete[]。下面我们就分析一下它们的实现原理。

无论 new 还是 new[]，C++ 必须知道返回指针所指向的内存块的大小，否则它就不可能正确地释放掉这块内存，这一点很像 C 语言中的 malloc。但是在用 new[] 为一个数组申请内存时，编译器还会悄悄地在内存中保存一个整数，用来表示数组中元素的个数。因为在 delete 一块内存时，我们不仅要知道指针指向多大的内存，更重要的是要知道指针指向的数组中对象的个数。因为只有知道了对象数量才能一一调用它们的析构函数，完成对数组中所有对象的清理。如果使用的是 delete，则编译器只会将指针所指的对象当作单个对象来处理。所以对于数组，需要使用 delete[] 来处理；符号 [] 会告诉编译器在 delete 这块内存时，先去获取保存的那个元素数量值，然后再进行一一清理。如果你对汇编有所了解，那么你可以通过反汇编代码对此一探究竟。

也许你会认为 C++ 这么设计绝对是多此一举，因为单个对象只是对象数组的一个特例，无论是一个对象，还是对象数组，我们都对元素个数进行记录，这样也就不再需要两个版本的 new 和 delete 了。但是 C++ 之父之所以没有选择这么做，也许是为了坚持他认定的 C++ 设计风格和宗旨：决不多费一点力。殊不知，这么做的直接后果就是需要程序员付出更多的细心与努力。

需要注意的是，由于内置数据类型没有构造、析构函数，所以在针对内置数据类型时，释放内存使用 delete 或 delete[] 的效果是一样的。例如：

```
int *pArray = new int[10];
... // processing code
delete pArray; // 等同于 delete[] pArray;
```

虽然针对内置类型，delete 和 delete[] 都能正确地释放所申请的内存空间，但是如果申请的是一个数组，建议还是使用 delete[] 形式。

所以，使用 new 和 delete 的一个简单有效的原则就是：如果在调用 new 时使用了 []，则你在调用 delete 时也使用 []，如果在调用 new 的时候没有用 []，那么也不应该在调用时使用 []。new 和 delete、new[] 和 delete[] 必须对应着使用。

对于那些喜欢 typedef 的人，还有一点需要提醒。因为在这种情况下很容易出现 new[] 和 delete 的混用。如下面的代码片段所示：

```
typedef int scorers[LESSONS_NUM];
int *pScorer = new scorers;
```

这该使用哪一种形式的 delete 呢？如下所示。

```
delete pScorer; // Wrong!!!
delete[] pScorer; // Right
```

为了避免出现这样的错误，建议不要对数组类型做 typedef，或者采用 STL 中的 vector 代替数组。

请记住：

new 和 delete、new[] 和 delete[] 必须对应使用，否则会出现未定义行为，导致程序崩溃。

建议 29：区分 new 的三种形态

C++ 语言一直被认为是复杂编程语言中的杰出代表之一，不仅仅是因为其繁缛的语法规则，还因为其晦涩的术语。下面要讲的就是你的老熟人——new：

它是一个内存管理的操作符，能够从堆中划分一块区域，自动调用构造函数，动态地创建某种特定类型的数据，最后返回该区域的指针。该数据使用完后，应调用 delete 运算符，释放动态申请的这块内存。

如果这就是你对 new 的所有认识，那么我不得不说，你依旧被 new 的和善外表所蒙蔽着。看似简单的 new 其实有着三种不同的外衣。

是的，你没有看错，也不用感到惊奇，一个简单的 new 确实有三种不同的形态，它扮演着三种不同的角色，如下所示：

- ❑ new operator
- ❑ operator new
- ❑ placement new

下面的代码片段展示的是我们印象中熟悉的那个 new：

```
string *pStr = new string("Memory Management");
int *pInt = new int(2011);
```

这里所使用的 new 是它的第一种形态 new operator。它与 sizeof 有几分类似，它是语言内建的，不能重载，也不能改变其行为，无论何时何地它所做的有且只有以下三件事，如图 3-2 所示。

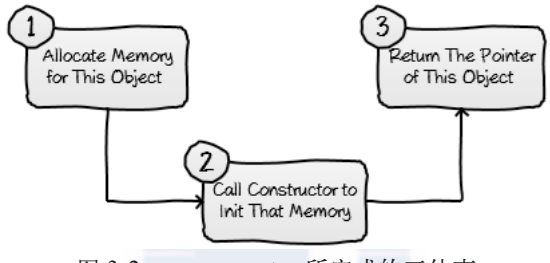

图 3-2 new operator 所完成的三件事

所以当写出“string *pStr = new string("Memory Management");”代码时，它其实做的就是以下几件事：

```
// 为 string 对象分配 raw 内存
void *memory = operator new( sizeof(string) );
// 调用构造函数，初始化内存中的对象
call string::string()on memory;
// 获得对象指针
string *pStr = static_cast<string*>(memory);
当然，对于内置类型，第二步是被忽略的，即：
// 为 int 分配 raw 内存
void *memory = operator new( sizeof(int) );
// 获得对象指针
int *pInt = static_cast<int*>(memory);
```

其实 new operator 背后还藏着一个秘密，即它在执行过程中，与其余的两种形态都发生了密切的关系：第一步的内存申请是通过 operator new 完成的；而在第二步中，关于调用什么构造函数，则由 new 的另外一种形态 placement new 来决定的。

对于 new 的第二种形态——内存申请中所调用的 operator new，它只是一个长着“明星脸”的普通运算符，具有和加减乘除操作符一样的地位，因此它也是可以重载的。

operator new 在默认情况下首先会调用分配内存的代码，尝试从堆上得到一段空间，同时它对事情的结果做了最充分的准备：如果成功则直接返回；否则，就转而去调用一个 new_hander，然后继续重复前面过程，直到异常抛出为止。所以如果 operator new 要返回，必须满足以下条件之一：

- 内存成功分配。
- 抛出 bad_alloc 异常。

通常，operator new 函数通过以下方式进行声明：

```
void* operator new(size_t size);
```

注意，这个函数的返回值类型是 void*，因为这个函数返回的是一个未经处理的指针，是一块未初始化的内存，它像极了 C 库中的 malloc 函数。如果你对这个过程不满意，那么可以通过重载 operator new 来进行必要的干预。例如：

```
class A
{
public:
    A(int a);
    ~A();
    void* operator new(size_t size);
    ...
};
void* A::operator new(size_t size)
{
    cout<<"Our operator new...");
    return ::operator new(size);
}
```

这里的 operator new 调用了全局的 new 来进行内存分配（::operator new(size)）。当然这里的全局 new 也是可以重载的，但是在全局空间中重载 void * operator new(size_t size) 函数将会改变所有默认的 operator new 的行为方式，所以必须十二分的注意。还有一点需要注意的是，正像 new 与 delete 一一对应一样，operator new 和 operator delete 也是一一对应的；如果重载了 operator new，那么也得重载对应的 operator delete。

最后，要介绍的是 new 的第三种形态——placement new。正如前面所说的那样，placement new 是用来实现定位构造的，可以通过它来选择合适的构造函数。虽然通常情况下，构造函数是由编译器自动调用的，但是不排除你有时确实想直接手动调用，比如对未初始化的内存进行处理，获取想要的对象，此时就得求助于一个叫做 placement new 的特殊的 operator new 了：

```
#include <new>
#include "ClassA.h"
int main()
```

```
{
    void *s = operator new(sizeof(A));
    A* p = (A*)s;
    new(p) A(2011);  //p->A::A(2011);
    ... // processing code
    return 0;
}
```

placement new 是标准 C++ 库的一部分，被声明在了头文件 <new> 中，所以只有包含了这个文件，我们才能使用它。它在 <new> 文件中的函数定义很简单，如下所示：

```
#ifndef __PLACEMENT_NEW_INLINE
#define __PLACEMENT_NEW_INLINE
inline void *__CRTDECL operator new(size_t, void *_Where) _THROW0()
{  // construct array with placement at _Where
   return (_Where);
}

inline void __CRTDECL operator delete(void *, void *) _THROW0()
{  // delete if placement new fails
}
 #endif /* __PLACEMENT_NEW_INLINE */
```

这就是 placement new 需要完成的事。细心的你可能会发现，placement new 的定义与 operator new 声明之间的区别：placement new 的定义多一个 void* 参数。使用它有一个前提，就是已经获得了指向内存的指针，因为只有这样我们才知道该把 placement new 初始化完成的对象放在哪里。

在使用 placement new 的过程中，我们看到的却是 "new(p) A(2011)" 这样奇怪的调用形式，它在特定的内存地址上用特定的构造函数实现了构造一个对象的功能，A(2011) 就是对构造函数 A(int a) 的显式调用。当然，如果显式地调用 placement new，那么也得本着负责任的态度显式地调用与之对应的 placement delete：p->~A();。这部分工作本来可以由编译器独自完成的：在使用 new operator 的时候，编译器会自动生成调用 placement new 的代码，相应的，在调用 delete operator 时同样会生成调用析构函数的代码。所以，除非特别必要，不要直接使用 placement new。但是要清楚，它是 new operator 的一个不可或缺的步骤。当默认的 new operator 对内存的管理不能满足我们的需要，希望自己手动管理内存时，placement new 就变得有用了。就像 STL 中的 allocator 一样，它借助 placement new 来实现更灵活有效的内存管理。

最后，总结一下：

- ❑ 如果是在堆上建立对象，那么应该使用 new operator，它会为你提供最为周全的服务。
- ❑ 如果仅仅是分配内存，那么应该调用 operator new，但初始化不在它的工作职责之内。如果你对默认的内存分配过程不满意，想单独定制，重载 operator new 是不二选择。

- 如果想在一块已经获得的内存里建立一个对象，那就应该用 placement new。但是通常情况下不建议使用，除非是在某些对时间要求非常高的应用中，因为相对于其他两个步骤，选择合适的构造函数完成对象初始化是一个时间相对较长的过程。

请记住：

不要自信地认为自己对 new 很熟悉，要正确区分 new 所具有的三种不同形态，并能在合适的情形下选择合适的形态，以满足特定需求。

建议 30：new 内存失败后的正确处理

应该有很多的程序员对比尔 • 盖茨的这句话有所耳闻：

对于任何一个人而言，640KB 应当是足够的了。（640K ought to be enough for everybody.）

不幸的是，伟大的比尔 • 盖茨也失言了。随着硬件水平的发展，内存变得越来越大，但是似乎仍不能满足人们对内存日益增长的需求。所以呢，我们 C/C++ 程序员在写程序时也必须考虑一下内存申请失败时的处理方式。

通常，我们在使用 new 进行内存分配的时候，会采用以下的处理方式：

```
char *pStr = new string[SIZE];
if(pStr == NULL)
{
     ... // Error processing
     return false;
}
```

你能发现上述代码中存在的问题吗？这是一个隐蔽性极强的臭虫（Bug）。

我们沿用了 C 时代的良好传统：使用 malloc 等分配内存的函数时，一定要检查其返回值是否为“空指针”，并以此作为检查分配内存操作是否成功的依据，这种 Test-for-NULL 代码形式是一种良好的编程习惯，也是编写可靠程序所必需的。可是，这种完美的处理形式必须有一个前提：若 new 失败，其返回值必须是 NULL。只有这样才能保证上述看似“逻辑正确、风格良好”的代码可以正确运行。

那么 new 失败后编译器到底是怎么处理的？在很久之前，即 C++ 编译器的蛮荒时代，C++ 编译器保留了 C 编译器的处理方式：当 operator new 不能满足一个内存分配请求时，它返回一个 NULL 指针。这曾经是对 C 的 malloc 函数的合理扩展。然而，随着技术的发展，标准的更新，编译器具有了更强大的功能，类也被设计得更漂亮，新时代的 new 在申请内存失败时具备了新的处理方式：抛出一个 bad_alloc exception（异常）。所以，在新的标准里，上述 Test-for-NULL 处理方式不再被推荐和支持。

如果再回头看看本建议开头的代码片段，其中的 if (pStr == 0) 从良好的代码风格突然一下变成了毫无意义。在 C++ 里，如果 new 分配内存失败，默认是抛出异常。所以，如果分配成功，pStr == 0 就绝对不会成立；而如果分配失败了，也不会执行 if (pStr == 0)，因为分配失败时，new 就会抛出异常并跳过后面的代码。

为了更加明确地理解其中的玄机，首先看看相关声明：

```
namespace   std
{
    class   bad_alloc
    {
            //   ...
    };
}

//   new   and   delete
void   *operator   new(std::size_t)   throw(std::bad_alloc);
void   operator   delete(void   *)   throw();

//   array   new   and   delete
void   *operator   new[](std::size_t)   throw(std::bad_alloc);
void   operator   delete[](void   *)   throw();

//   placement   new   and   delete
void   *operator   new(std::size_t,   void   *)   throw();
void   operator   delete(void   *,   void   *)   throw();

//   placement   array   new   and   delete
void   *operator   new[](std::size_t,   void   *)   throw();
void   operator   delete[](void   *,   void   *)   throw();
```

在以上的 new 操作族中，只有负责内存申请的 operator new 才会抛出异常 std::bad_alloc。如果出现了这个异常，那就意味着内存耗尽，或者有其他原因导致内存分配失败。所以，按照 C++ 标准，如果想检查 new 是否成功，则应该捕捉异常：

```
try
{
   int* pStr = new string[SIZE];
   ...  // processing codes
}
catch ( const bad_alloc& e )
{
   return -1;
}
```

但是市面上还存在着一些古老编译器的踪迹，这些编译器并不支持这个标准。同时，在这个标准制定之前已经存在的很多代码，如果因为标准的改变而变得漏洞百出，肯定会引起很多人抗议。C++ 标准化委员会并不想遗弃这些 Test-for-NULL 的代码，所以他

们提供了 operator new 的另一种可选形式—— nothrow ，用以提供传统的 Failure-yields-NULL 行为。

其实现原理如下所示：

```
void * operator new(size_t cb, const std::nothrow_t&) throw()
{
  char *p;
  try
  {
     p = new char[cb];
  }
  catch (std::bad_alloc& e)
  {
     p = 0;
  }
  return p;
}
```

<new> 文件中也声明了 nothrow new 的重载版本，其声明方式如下所示：

```
namespace   std
{
    struct   nothrow_t
    {
          //   ...
    };
    extern   const   nothrow_t   nothrow;
}

//  new   and   delete
void   *operator   new(std::size_t,   std::nothrow_t   const   &)   throw();
void   operator   delete(void   *,   std::nothrow_t   const   &)   throw();

//   array   new   and   delete
void   *operator   new[](std::size_t,   std::nothrow_t   const   &)   throw();
void   operator   delete[](void   *,   std::nothrow_t   const   &)   throw();
```

如果采用不抛出异常的 new 形式，本建议开头的代码片段就应该改写为以下形式：

```
int* pStr = new(std::nothrow) string[SIZE];
if(pStr==NULL)
{
   ...  // 错误处理代码
}
```

根据建议 29 可知，编译器在表达式 new (std::nothrow) ClassName 中一共完成了两项任务。首先，operator new 的 nothrow 版本被调用来为一个 ClassName object 分配对象内存。假如这个分配失败，operator new 返回 null 指针；假如内存分配成功，ClassName 的构造函

数则被调用，而在此刻，对象的构造函数就能做任何它想做的事了。如果此时它也需要 new 一些内存，但是没有使用 nothrow new 形式，那么，虽然在 "new (std::nothrow) ClassName" 中调用的 operator new 不会抛出异常，但其构造函数却无意中办了件错事。假如它真的这样做了，exception 就会像被普通的 operator new 抛出的异常一样在系统里传播。所以使用 nothrow new 只能保证 operator new 不会抛出异常，无法保证 "new (std::nothrow) ClassName" 这样的表达式不会抛出 exception。所以，慎用 nothrow new。

最后还需要说明一个比较特殊但是确实存在的问题：在 Visual C++ 6.0 中目前 operator new、operator new(std::nothrow) 和 STL 之间不兼容、不匹配，而且不能完全被修复。如果在非 MFC 项目中使用 Visual C++6.0 中的 STL，其即装即用的行为可能导致 STL 在内存不足的情况下让应用程序崩溃。对于基于 MFC 的项目，STL 是否能够幸免于难，完全取决于你使用的 STL 针对 operator new 的异常处理。这一点，在 James Hebben 的文章《不要让内存分配失败导致您的旧版 STL 应用程序崩溃》中进行了详细的介绍，如果你在使用古老的 Visual C++ 6.0 编译器，而且对这个问题充满兴趣，请 Google 之。

请记住：

当使用 new 申请一块内存失败时，抛出异常 std::bad_alloc 是 C++ 标准中规定的标准行为，所以推荐使用 try{ p = new int[SIZE]; } catch(std::bad_alloc) { ... } 的处理方式。但是在一些老旧的编译器中，却不支持该标准，它会返回 NULL，此时具有 C 传统的 Test_for_NULL 代码形式便起了作用。所以，要针对不同的情形采取合理的处置方式。

建议 31：了解 new_handler 的所作所为

在使用 operator new 申请内存失败后，编译器并不是不做任何的努力直接抛出 std::alloc 异常，在这之前，它会调用一个错误处理函数（这个函数被称为 new-handler），进行相应的处理。通常，一个好的 new-handler 函数的处理方式必须遵循以下策略之一：

❑ Make more memory available（使更大块内存有效）

operator new 会进行多次的内存分配尝试，这可能会使其下一次的内存分配尝试成功。其中的一个实现方法是在程序启动时分配一大块内存，然后在 new-handler 第一次被调用时释放它供程序使用。

❑ Install a different new-handler（装载另外的 new-handler）

程序中可以同时存在多个 new-handler，假如当前的 new-handler 不能获得更多的内存供 operator new 分配使用，但另一个 new-handler 却可以做到。在这种情形下，当前的 new-handler 则会通过调用 set_new_handler 在它自己的位置上安装另一个 new-handler。当 operator new 下一次调用 new-handler 时，它会调用最新安装的那一个。

❑ Deinstall the new-handler（卸载 new-handler）

换句话说，就是将空指针传给 set_new_handler，此时就没有了相应的 new-handler。当内存分配失败时，operator new 则会抛出一个异常。

❑ Throw an exception（抛出异常）

抛出一个类型为 bad_alloc 或继承自 bad_alloc 的其他类型的异常。

❑ Not return（无返回）

直接调用 abort 或 exit 结束应用程序。

以上的这些处理方式让我们在实现 new-handler functions 时拥有了更多的选择与自由。这些各式各样的 new-handler 函数是可以通过调用标准库函数 set_new_handler 进行特殊定制的，你可以按照自己的方式来对编译器的这一行为进行设定。这个函数同样也声明在 <new> 中：

```
namespace std
{
  typedef void (*new_handler)();
  new_handler set_new_handler(new_handler p) throw();
}
```

通过函数声明可以看到 set_new_handler 的形参是一个指向函数的指针，这个函数在 operator new 无法分配被请求的内存时调用。set_new_handler 的返回值是一个指向函数的指针，指向的是 set_new_handler 调用之前的异常处理函数。所以，可以按照以下方式使用 set_new_handler 函数：

```
//error-handling function
void MemErrorHandling()
{
  std::cerr << "Failed to allocate memory\n";
  std::abort();
}
//Application
const long long DATA_SIZE = 1024*1024*1024;
int main()
{
    std::set_new_handler(MemErrorHandling);
    std::cout << "Attempting to allocate 1 GB...";
    char *pDataBlock = NULL;
    try
    {
          pDataBlock = new char[DATA_SIZE];
    }
    catch(std::alloc& e)
    {
          ... //some processing codes
    }
```

```
    ... // other processing code
}
```

假如operator new分配空间的请求得不到满足，MemErrorHandling函数将被调用，程序将按照函数中的设定处理方式运行。在标准C++中，标准set_new_handler为用户类统一指定了错误处理函数global new-handler。上述代码采用的就是global new-handler形式。

通过上述示例代码可以看出，new_handler必须有主动退出的功能，否则就会导致operator new内部死循环。因此new_handler一般会采用如下形式，伪代码表示如下：

```
void MemErrorHandling()
{
   if( 有可能使得operator new成功 )
   {
       做有可能使得operator new成功的事
       return;
   }
   // 主动退出
    abort/exit 直接退出程序
    或 set_new_handler( 其他newhandler);
    或 set_new_handler(0)
    或 throw bad_alloc() 或派生类
}
```

当然，我们可以根据被分配对象的不同，采用不同的方法对内存分配失败进行处理，实现对class-specific new-handlers的支持。为了实现这一行为，需要为每一个class提供专属的set_new_handler和operator new版本。假设要为A class设定特殊的内存分配失败处理方式，则需要在类A中声明一个new_handler类型的静态成员（static member），并将其设置为A class的new-handler处理函数。所以就得到了下面的代码：

```
class A
{
public:
   static std::new_handler set_new_handler(std::new_handler p) throw();
   static void * operator new(std::size_t size) throw(std::bad_alloc);
   static void MemoryErrorHandling();
private:
   static std::new_handler m_curHandler;
};
// 静态类成员定义
std::new_handler A::m_curHandler = NULL;
```

C++标准中规定set_new_handler函数应该保存传递给它的函数指针，并返回前次调用时被保存的函数指针。上面A类中的set_new_handler也应该这么做：

```
std::new_handler A::set_new_handler(std::new_handler p) throw()
```

```
{
   std::new_handler oldHandler = m_curHandler;
   m_curHandler = p;
   return oldHandler;
}
```

接下来，我们自己定义该类的处理函数：

```
void MemoryErrorHandling()
{
   ... // processing code
}
void * operator new(std::size_t size) throw(std::bad_alloc)
{
    set_new_handler(MemoryErrorHandling);
    return ::operator new(size);
}
```

当然我们可以采用更好的设计方式（如类继承）来实现，这里就不赘述。如果读者感兴趣可以自己思考一下，或者求助于资源丰富的 Internet，它会提供详尽的参考资料。

请记住：

了解 new_handler 的所作所为，并通过标准库函数 set_new_handler 对内存分配请求不能被满足的处理函数进行特殊定制。

建议 32：借助工具监测内存泄漏问题

内存管理确实是一个令众多 C/C++ 程序员感到费神又费力的问题，内存错误通常都具有隐蔽性，难以再现，而且其症状一般不能在相应的源代码中找到。C/C++ 应用程序的大部分缺陷和错误都和内存相关，预防、发现、消除代码中和内存相关的缺陷，成为 C/C++ 程序员编写、调试、维护代码时的重要任务。然而任何人都无法时刻高度谨慎，百密中难免会有一疏，一不小心就会发生内存问题。如果泄漏内存，则运行速度会逐渐变慢，并最终会停止运行；如果覆盖内存，则程序会变得非常脆弱，很容易受到恶意用户的攻击。因此，需要特别关注 C/C++ 编程的内存问题，特别是内存泄漏。幸运的是，现在有许多的技术和工具能够帮助我们验证内存泄漏是否存在，寻找到发生问题的位置。

内存泄漏一般指的是堆内存的泄漏。如果我们使用 malloc 函数或 new 操作符从堆中分配到一块内存，在使用完后，程序员必须负责调用相应的 free 或 delete 显式地释放该内存块，否则，这块内存就不能被再次使用，此时就出现了传说中的“内存泄漏”问题。如下面的代码片段所示：

```
void Function(size_t nSize)
{
    char* pChar= new char[nSize];
    if( !SetContent(pChar, nSize ) )
    {
         cout<<"Error: Fail To Set Content"<<endl;
         return;
    }
    ...//using pChar
    delete pChar;
}
```

程序在入口处分配内存，在出口处释放内存，但是这里忽视了代码片段中的 return;，如果函数 SetContent() 失败，指针 pChar 指向的内存就不会被释放，会发生内存泄漏。这是一种常见的内存泄漏情形。

检测内存泄漏的关键是要能截获对分配内存和释放内存的函数的调用。通过截获的这两个函数，我们就能跟踪每一块内存的生命周期。每当成功分配一块内存时，就把它的指针加入一个全局的内存链中；每当释放一块内存时，再把它的指针从内存链中删除。这样，当程序运行结束的时候，内存链中剩余的指针就会指向那些没有被释放的内存。这就是检测内存泄漏的基本原理[⊖]。

检测内存泄漏的常用方法有如下几种：

❑ MS C-Runtime Library 内建的检测功能

使用 MFC 开发的应用程序时，会在 Debug 模式下编译执行，程序运行结束后，Visual C++ 会输出内存的使用情况，如果发生了内存泄漏，在 Debug 窗口中会输出所有发生泄漏的内存块的信息，如下所示：

```
Detected memory leaks!
Dumping objects ->
mainFrm.cpp(45) : {352} normal block at 0x0058A4B8, 40 bytes long.
Data: <                > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
Object dump complete.
```

这是因为在编译过程中，IDE 自动加入了内存泄漏的检测代码。MFC 在程序执行过程中维护了一个内存链，以便跟踪每一块内存的生命周期。在程序退出的时候，dbgheap.c 文件中的 extern "C" _CRTIMP int __cdecl _CrtDumpMemoryLeaks(void) 函数被调用，遍历当前的内存链，如果发现存在没有被释放的内存，则打印出内存泄露的信息。

一般，大家都误以为这些内存泄漏的检测功能是由 MFC 提供的，其实不然。这是 VC++ 的 C 运行库（CRT）提供的功能，MFC 只是封装和利用了 MS C-Runtime Library 的

⊖ 详细的算法可以参见 Steve Maguire 的《Writing Solid Code》。

Debug Function 而已。所以，在编写非 MFC 程序时我们也可以利用 MS C-Runtime Library 的 Debug Function 加入内存泄漏的检测功能。

要在非 MFC 程序中打开内存泄漏的检测功能非常容易，只须在程序的入口处添加以下代码：

```
_CrtSetDbgFlag(  _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG)
               |  _CRTDBG_LEAK_CHECK_DF               );
```

这样，在程序运行结束时，如果还有内存块没有释放，它们的信息就会被打印到 Debug 窗口里，如下面的代码片段所示：

```
#include <crtdbg.h>

#ifdef _DEBUG
#define new   new(_NORMAL_BLOCK, __FILE__, __LINE__)
#endif
void EnableMemLeakCheck()
{
    _CrtSetDbgFlag( _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG)
                  | _CRTDBG_LEAK_CHECK_DF              );
}

int main()
{
    EnableMemLeakCheck();
    _CrtSetBreakAlloc(53);
    int* pLeak = new int[10];

    return 0;
}
```

在 Debug 模式下，程序退出时，内存块 pLeak 因为没有显式地释放，发生了内存泄漏，泄漏信息被打印出来：

```
Detected memory leaks!
Dumping objects ->
main.cpp(26) : {53} normal block at 0x002E1508, 40 bytes long.
 Data: <                > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
 Object dump complete.
```

请读者思考一下，_CrtSetBreakAlloc(53) 起到的是什么作用？

目前这种方式只支持 MS 系统开发环境。当然，如果开发系统环境是 Linux，也可以根据 MS C-Runtime Library 内建检测功能的实现方式开发出自己的 Linux C-Runtime Library 内建检测版本。

❑ 外挂式的检测工具

如果开发的是一个大型程序，MS C-Runtime Library 提供的检测功能便显得有点笨拙了。

此时，我们可以采用外挂式的检测工具 BoundsChecker 或 Insure++。

BoundsChecker 采用的是一种被称为 Code Injection 的技术，来截获对分配内存和释放内存的函数的调用的。简单地说，当程序开始运行时，BoundsChecker 的 DLL 被自动载入进程的地址空间中，然后它会修改进程中对内存分配和释放的函数调用，让这些调用首先转入它的代码，然后再执行原来的代码。BoundsChecker 在做这些动作时，无须修改被调试程序的源代码或工程配置文件，这使得使用它非常简便、直接。而 Insure++ 则是利用其专利技术（源码插装和运行时指针跟踪）来发现大量的内存操作错误，准确报告错误的源代码行和执行轨迹。

如果开发环境是 Linux，MS C-Runtime Library 内建检测功能就会彻底失效，BoundsChecker 或 Insure++ 也无能为力。这时，外挂式的检测工具 Rational Purify 或 Valgrind 便派上了用场。

Rational Purify 主要是针对软件开发过程中难以发现的内存错误、运行时错误。它可以在软件开发过程中自动地发现错误，准确地定位错误，并提供完备的错误信息，从而减少调试时间。同时它也是市场上唯一支持多种平台的相关工具，并且可以和很多主流开发工具集成。Purify 可以检查应用的每一个模块，甚至可以查出复杂的多线程或进程应用中的错误。另外，它不仅可以检查 C/C++，还可以对 Java 或 .NET 中的内存泄漏问题给出报告。

在 Linux 系统中，使用 Purify 非常简单，只须重新编译程序：

```
purify g++ -g main.cpp -o LeakDetector
```

运行编译生成的可执行文件 LeakDetector，就可以定位出内存泄漏的具体位置。

除了 Rational Purify，Valgrind 也是 Linux 系统下开发应用程序时用于调试内存问题的有效工具。它尤其擅长发现内存管理的问题，检查发现程序运行时的内存泄漏。

至于上述这些外挂式检测工具的具体使用方法就不赘述了。

根据应用程序的具体情况，合理采用上述方法和工具，可以有效防止和查找代码中的内存泄漏问题，并且能和开发人员日常编码无缝结合，有效提高开发效率，增强应用程序鲁棒性。

请记住：

内存泄露是一个大问题，但是可以通过一定的方法或借助于专业的检测工具，来查找并发现这些问题，有效地提升程序员的开发效率。

建议 33：小心翼翼地重载 operator new/ operator delete

虽然 C++ 标准库已经为我们提供了 new 与 delete 操作符的标准实现，但是由于缺乏对具体对象的具体分析，系统默认提供的分配器在时间和空间两方面都存在着一些问题 ：分配器速度较慢，而且在分配小型对象时空间浪费比较严重，特别是在一些对效率或内存有较大

限制的特殊应用中。比如说在嵌入式的系统中，由于内存限制，频繁地进行不定大小的内存动态分配很可能会引起严重问题，甚至出现堆破碎的风险；再比如在游戏设计中，效率绝对是一个必须要考虑的问题，而标准 new 与 delete 操作符的实现却存在着天生的效率缺陷。此时，我们可以求助于 new 与 delete 操作符的重载，它们给程序带来更灵活的内存分配控制。除了改善效率，重载 new 与 delete 还可能存在以下两点原因：

- 检测代码中的内存错误。
- 获得内存使用的统计数据。

相对于其他的操作符，operator new 具有一定的特殊性，在多个方面上与它们大不相同。首先，对于用户自定义类型，如果不重载，其他操作符是无法使用的，而 operator new 则不然，即使不重载，亦可用于用户自定义类型。其次，在参数方面，重载其他操作符时参数的个数必须是固定的，而 operator new 的参数个数却可以是任意的，只需要保证第一个参数为 size_t 类型，返回类型为 void * 类型即可。所以 operator new 的重载会给我们一种错觉：它更像是一个函数重载，而不是一个操作符重载。

关于 operator new 重载函数的形式，在 C++ 标准中有如下规定：

分配函数应当是一个类的成员函数或者是全局函数；如果一个分配函数被放于非全局名空间中，或者是在全局名空间被声明为静态，那这个程序就是格式错误的。⊖

也就是说，重载的 operator new 必须是类成员函数或全局函数，而不可以是某一名空间之内的函数或是全局静态函数。此外，还要多加注意的是，重载 operator new 时需要兼容默认的 operator new 的错误处理方式，并且要满足 C++ 的标准规定：当要求的内存大小为 0 byte 时也应该返回有效的内存地址。

所以，全局的 operator new 重载应该不改变原有签名，而是直接无缝替换系统原有版本，如下所示：

```
void * operator new(size_t size)
{
    if(size == 0)
       size = 1;
    void *res;
    for(;;)
    {
        //allocate memory block
        res = heap_alloc(size);
        //if successful allocation, return pointer to memory
        if(res)
            break;
```

⊖ An allocation function shall be a class member function or a global function; a program is ill-formed if an allocation function is declared in a namespace scope other than global scope or declared static in global scope.

```
        //call installed new handler
        if (!CallNewHandler(size))
            break;
        //new handler was successful -- try to allocate again
    }
    return res;
}
```

如果是用这种方式进行的重载，再使用时就不需要包含 new 头文件了。“性能优化”时通常采用这种方式。

如果重载了一个 operator new，记得一定要在相同的范围内重载 operator delete。因为你分配出来的内存只有你自己才知道应该如何释放。如果你偷懒或者是忘记了，编译器就会求助于默认的 operator delete，用默认方式释放内存。虽然程序编译可以通过，但是这将导致惨重的代价。所以，你必须时刻记得在写下 operator new 的同时写下 operator delete。相对于 operator new，重载 operator delete 要简单许多，如下所示：

```
void operator delete(void* p)
{
     if(p==NULL)
        return;
     free(p);
}
```

唯一要注意的一点就是，须遵循 C++ 标准中要求删除一个 NULL 指针是安全的这一规定。在全局空间中重载 void * operator new(size_t size) 函数将会改变所有默认的 operator new 的行为方式，所以一定要小心使用。

如果使用不同的参数类型重载 operator new/delete，则请采用如下函数声明形式：

```
//返回的指针必须能被普通的 ::operator delete(void*) 释放
void* operator new(size_t size, const char* file, int line);
//析构函数抛异常时被调用
void operator delete(void* p, const char* file, int line);
```

调用时采用以下方式：

```
string* pStr = new (__FILE, __LINE__) string;
```

这样就能跟踪内存分配的具体位置，定位这个动作发生在哪个文件的哪一行代码中了。在“检测内存错误”和“统计内存使用数据”时通常会用这种方式重载。

此外，我们还可以为 operator new 的重载使用参数默认值，甚至是不定参数。其原则和普通函数重载一样。

但是在使用全局重载时应该慎之又慎，因为这样做非常具有侵略性：这会让使用你编写的库的人没有选择的余地；同时，如果两个 lib 中都对 operator new 进行了重载，在使用时会出现这样的错误：duplicated symbol link error。这是多么令人恼火的一件事啊。

与全局 ::operator new() 不同，具体类的 operator new 与 delete 的影响面要小得多，它只影响本 class 及其派生类。为某个 class 重载 operator new 时必须将其定义为类的静态函数。因为 operator new 是在类的具体对象被构建出来之前调用的，在调用 operator new 的时候 this 指针尚未诞生，因此重载的 operator new 必须是 static 的：

```
class B
{
public:
    static void * operator new(size_t size);
    static void operator delete(void *p);
    // other members
};
void *B::operator new(size_t size)
{
   ...
}
void B::operator delete(void *p)
{
   ...
}
```

当然，同全局 operator new 重载一样，在类中重载成员 operator new 也可以添加额外的参数，并且可以使用默认值。另外，成员 operator new 也是可以继承的。但类中的 operator delete 也必须声明为静态函数。因为调用 operator delete 时，对象已经被析构，this 指针业已灰飞烟灭。

虽然为单独的 class 重载成员 operator new/ delete 是可行的，但不推荐使用。因为既然对它们进行了重载，说明它的内存分配策略已被进行了精心的特殊定制，从类似 ClassName * p =new ClassName 形式的代码中我们根本不能获得此信息。而且，我们有更加简单明了的 Factory 方案可以使用：

```
static ClassName* ClassName::CreateObject();
```

清晰明确优于模糊不清（Explicit is better than implicit），对此我深信不疑。

关于 operator new/operator delete 的重载，还有一个必须小心的问题，那就是在内存分配机制中必须要考虑对象数组内存分配这一点。C++ 将对象数组的内存分配看作是一个不同于单个对象内存分配的单独操作。对于多数的 C++ 实现，因为需要额外存储对象数量，new[] 操作符中的个数参数会是数组的大小加上存储对象数目的一些字节。所以，如果希望改变对象数组的分配方式，同样需要重载 new[] 和 delete[] 操作符。

请记住：

通过重载 operator new 和 operator delete 的方法，可以自由地采用不同的分配策略，从不同的内存池中分配不同的类对象。但是是否选择重载 operator new/delete 一定要深思熟虑。

建议 34：用智能指针管理通过 new 创建的对象

前面的建议中我们不厌其烦的一再重复：内存泄漏是一个很大很大的问题！为了应对这个问题，已经有许多技术被研究出来，比如 Garbage Collection（垃圾回收）、Smart Pointer（智能指针）等。Garbage Collection 技术一直颇受注目，并且在 Java 中已经发展成熟，成为内存管理的一大利器，但它在 C++ 语言中的发展却不顺利，C++ 为了追求运行速度，20 年来态度坚决地将其排除在标准之外。真不知 C++ 通过加大开发难度来换取执行速度的做法究竟是利还是弊。为了稍许平复因为没有 Garbage Collection 而引发的 C++ 程序员的怨气，C++ 对 Smart Pointer 技术采取了不同的态度，它选择对这一技术的支持，并在 STL 中包含了支持 Smart Pointer 技术的 class，赐予了 C/C++ 程序员们一件管理内存的神器。

Smart Pointer 是 Stroustrup 博士所推崇的 RAII（Resource Acquisition In Initialization）的最好体现。该方法使用一个指针类来代表对资源的管理逻辑，并将指向资源的句柄（指针或引用）通过构造函数传递给该类。当离开当前范围（scope）时，该对象的析构函数一定会被调用，所以嵌在析构函数中的资源回收的代码也总是会被执行。这种方法的好处在于，由于将资源回收的逻辑通过特定的类从原代码中剥离出来，自动正确地销毁动态分配的对象，这会让思路变得更加清晰，同时确保内存不发生泄露。

它的一种通用实现技术是使用引用计数（Reference Count）。引用计数智能指针，是一种生命期受管的对象，其内部有一个引用计数器。当内部引用计数为零时，这些对象会自动销毁自身的智能指针类。每次创建类的新对象时，会初始化指针并将引用计数置为 1；当对象作为另一对象的副本而创建时，它会调用拷贝构造函数拷贝指针并增加与之相应的引用计数；对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数；如果引用计数减至 0，则删除对象，并增加右操作数所指对象的引用计数；调用析构函数时，构造函数减少引用计数，直到计数为 0，释放对象空间。

Smart Pointer 具有非常强大的能力，谨慎而明智的选择能给我们带来极大的便利。前面已经说到 STL 中包含了支持 Smart Pointer 技术的 class，它就是智能指针：auto_ptr。要使用 auto_prt，首先要包含 memory 头文件：

```
#include <memory>
```

auto_ptr 可以指向一个以 new 建立的对象，当 auto_ptr 的生命周期结束时，其所指向的对象之资源也会被自动释放，且不必显式地调用 delete，而对象指针的操作依旧如故。例如：

```
class A
{
public:
    A(){}
    ~A(){}
    void Hello()
```

```
    {
      std::cout<<"Hello Smart Pointer";
    }
};

int main()
{
    std::auto_ptr<A> pA(new A());
    pA->Hello();
    return 0;
}
```

当然，也可以建立一个未指向任何对象的 auto_prt，例如：

```
std::auto_ptr<int> iPtr;
```

它就像空指针，未指向任何对象，所以也就不能进行操作，但是可以通过 get() 函数来判断它是否指向对象的地址：

```
if(iPtr.get() == 0) // 不指向任何对象
{
    iPtr.reset(new int(2011)); // 指向一个对象
}
```

auto_ptr 还可以使用另一个 auto_ptr 来建立，但是需要十分小心的是，这会造成所有权的转移，例如：

```
auto_ptr< string> sPtr1 (new string("Smart Pointer"));
auto_ptr< string> sPtr2 (sPtr1);
if( !sPtr1->empty() )
  cout<<*sPtr1<< endl;
```

当使用 sPtr1 来建立 sPtr2 时，sPtr1 不再对所指向对象的资源释放负责，而是将接力棒传递到了 sPtr2 的手里，sPtr1 丧失了使用 string 类成员函数的权利，所以在判断 sPtr1->empty() 时程序会崩溃。

auto_ptr 的资源维护动作是以 inline 的方式来完成的，在编译时代码会被扩展开来，所以使用它并不会牺牲效率。虽然 auto_ptr 指针是一个 RAII 对象，能够给我们带来很多便利，但是它的缺点同样不可小觑：

- auto_ptr 对象不可作为 STL 容器的元素，所以二者带来的便利不能同时拥有。这一重大缺陷让 STL 的忠实拥趸们愤怒不已。
- auto_ptr 缺少对动态配置而来的数组的支持，如果用它来管理这些数组，结果是可怕的、不可预期的。
- auto_ptr 在被复制的时候会发生所有权转移。

Smart Pointer 作为 C++ 垃圾回收机制的核心，必须足够强大、具有工业强度，并且保证

安全性。可是 STL 中的 auto_ptr 却像是扶不起的阿斗，不堪大用。在这样的情况下，C++ 标准委员会自然需要考虑引入新的智能指针。其中由 C++ 标准委员会库工作组发起的 Boost 组织开发的 Boost 系列智能指针最为著名。除此之外，还有 Loki 库提供的 SmartPtr、ATL 提供的 CComPtr 和 CComQIPtr。一个好消息是，就在 2011 年的 9 月刚刚获得通过的 C++ 新标准 C++ 11 中废弃了 auto_ptr 指针，取而代之的是两个新的指针类：shared_ptr 和 unique_ptr。shared_ptr 只是单纯的引用计数指针，unique_ptr 是用来取代 auto_ptr 的。unique_ptr 提供了 auto_ptr 的大部分特性，唯一的例外是 auto_ptr 的不安全、隐性的左值搬移；而 unique_ptr 可以存放在 C++0x 提出的那些能察觉搬移动作的容器之中。

在 Boost 中的智能指针共有五种：scoped_ptr、scoped_array、shared_ptr、shared_array、weak_ptr，其中最有用的就是 shared_ptr，它采取了引用计数，并且是线程安全的，同时支持扩展，推荐在大多数情况下使用。

boost::shared_ptr 支持 STL 容器：

```
typedef boost::shared_ptr<string>  CStringPtr;
std::vector< CStringPtr > strVec;
strVec.push_back( CStringPtr(new string("Hello")) );
```

当 vector 被销毁时，其元素——智能指针对象才会被销毁，除非这个对象被其他的智能指针引用，如下面的代码片段所示：

```
typedef boost::shared_ptr<string>  CStringPtr;
std::vector< CStringPtr > strVec;
strVec.push_back( CStringPtr(new string("Hello")) );
strVec.push_back( CStringPtr(new string("Smart")) );
strVec.push_back( CStringPtr(new string("Pointer")) );

CStringPtr strPtr = strVec[0];
strVec.clear(); //strVec 清空，但是保留了 strPtr 引用的 strVec[0]
cout<<*strPtr<<endl; // strVec[0] 依然有效
```

Boost 智能指针同样支持数组，boost::scoped_array 和 boost::shared_array 对象指向的是动态配置的数组。

Boost 的智能指针虽然增强了安全性，处理了潜在的危险，但是我们在使用时还是应该遵守一定的规则，以确保代码更加鲁棒。

规则 1：Smart_ptr<T> 不同于 T*

Smart_ptr<T> 的真实身份其实是一个对象，一个管理动态配置对象的对象，而 T* 是指向 T 类型对象的一个指针，所以不能盲目地将一个 T* 和一个智能指针类型 Smart_ptr<T> 相互转换。

- ❑ 在创建一个智能指针的时候需要明确写出 Smart_ptr<T> tPtr<new T>。
- ❑ 禁止将 T* 赋值给一个智能指针。

❑ 不能采用 tPtr = NULL 的方式将 tPtr 置空，应该使用智能指针类的成员函数。

规则 2：不要使用临时的 share_ptr 对象

如下所示：

```
class A;
bool IsAllReady();
void ProcessObject(boost::shared_ptr< A> pA,  bool isReady);
ProcessObject(boost::shared_ptr(new A), IsAllReady());
```

调用 ProcessObject 函数之前，C++ 编译器必须完成三件事：

（1）执行 "new A"。

（2）调用 boost::shared_ptr 的构造函数。

（3）调用函数 IsAllReady()。

因为函数参数求值顺序的不确定性，如果调用 IsAllReady() 发生在另外两个过程中间，而它又正好出现了异常，那么 new A 得到的内存返回的指针就会丢失，进而发生内存泄露，因为返回的指针没有被存入我们期望能阻止资源泄漏的 boost::shared_ptr 上。避免出现这种问题的方式就是不要使用临时的 share_ptr 对象，改用一个局部变量来实现，在一个独立的语句中将通过 new 创建出来的对象存入智能指针中：

```
boost::shared_ptr<A> pA(new A)
ProcessObject(pA, IsAllReady());
```

如果疏忽了这一点，当异常发生时，可能会引起微妙的资源泄漏。

请记住：

时刻谨记 RAII 原则，使用智能指针协助我们管理动态配置的内存能给我们带来极大的便利，但是需要我们谨慎而明智地做出选择。

建议 35：使用内存池技术提高内存申请效率与性能

Doug Lea 曾有言曰：“自 1960 年以来，动态内存分配就已经成为大多计算机系统的重要部分。”[⊖]

动态内存管理确实是件让人头疼的事儿，然而在实际的编程实践中，又不可避免地要大量用到堆上的内存。而这些通过 malloc 或 new 进行的内存分配却有着一些天生的缺陷：一方面，利用默认的内存管理函数在堆上分配和释放内存会有一些额外的开销，需要花费很多时间；另一方面，也是更糟糕的，随着时间的流逝，内存将形成碎片，一个应用程序的运行会越来越慢。

⊖ Dynamic memory allocation has been a fundamental part of most computer systems since roughly 1960... 摘自 Doug Lea 所写文章《A Memory Allocator》，详见 http://gee.cs.oswego.edu/dl/html/malloc.html。

当程序中需要对相同大小的对象频繁申请内存时，常会采用内存池（Memory Pool）技术来提高内存申请效率。经典的内存池技术，是一种用于分配大量大小相同的小对象的技术。通过该技术可以极大地加快内存分配 / 释放过程。内存池技术通过批量申请内存，降低了内存申请次数，从而节省了时间。对于大批量的小对象而言，使用内存池技术整体申请内存，减少了内存碎片的产生，对性能提升的帮助也是很显著的。

内存池技术的基本原理通过这个“池”字就进行了很好的自我阐释：应用程序可以通过系统的内存分配调用预先一次性申请适当大小的内存块（Block），并会将它分成较小的块（Smaller Chunks），之后每次应用程序会从先前已经分配的块（chunks）中得到相应的内存空间，对象分配和释放的操作都可以通过这个“池”来完成。只有当“池”的剩余空间太小，不能满足应用程序需要时，应用程序才会再调用系统的内存分配函数对其大小进行动态扩展。

经典的内存池实现原理如下：

```
class MemPool
{
public:
  MemPool(int nItemSize, int nMemBlockSize = 2048)
     : m_nItemSize(nItemSize),
     m_nMemBlockSize(nMemBlockSize),
     m_pMemBlockHeader(NULL),
     m_pFreeNodeHeader(NULL)
   {
   }
   ~ MemPool();
   void* Alloc();
   void  Free();
private:
    const int m_nMemBlockSize;
    const int m_nItemSize;

    struct _FreeNode
    {
        _FreeNode* pPrev;
        BYTE data[m_nItemSize - sizeof(_FreeNode*)];
    };

    struct _MemBlock
    {
        _MemBlock* pPrev;
        _FreeNode data[m_nMemBlockSize/m_nItemSize];
    };

    _MemBlock* m_pMemBlockHeader;
    _FreeNode* m_pFreeNodeHeader;
};
```

其中 MemPool 涉及两个常量：m_nMemBlockSize、m_nItemSize，还有两个指针变量 m_pMemBlockHeader、m_pFreeNodeHeader。指针变量 m_pMemBlockHeader 是用来把所有申请的内存块（MemBlock）串成一个链表。m_pFreeNodeHeader 变量则是把所有自由的内存结点（FreeNode）串成一个链表。内存块在申请之初就被划分为了多个内存结点，每个结点的大小为 ItemSize（对象的大小），共计 MemBlockSize/ItemSize 个。然后，这些内存结点会被串成链表。每次分配的时候从链表中取一个给用户，不够时继续向系统申请大块内存。在释放内存时，只须把要释放的结点添加到自由内存链表 m_pFreeNodeHeader 中即可。在 MemPool 对象析构时，可完成对内存的最终释放。

Boost 库同样对该技术提供了较好的支持：

❑ pool（#include <boost/pool/pool.hpp>）

boost::pool 用于快速分配同样大小的小块内存。如果无法分配，返回 0，如下所示。

```
boost::pool<> p(sizeof(double)); // 指定每次分配块的大小
if(p!=NULL)
{
    double* const d = (double*)p.malloc(); // 为 d 分配内存
    pA.free(d);   // 将内存还给 pool
}
```

pool 的析构函数会释放 pool 占用的内存。

❑ object_pool（#include <boost/pool/object_pool.hpp>）

object_pool 和 pool 的区别在于：pool 指定每次分配的块的大小，object_pool 指定分配的对象的类型，如下所示：

```
boost::object_pool<A> p;
```

用 A * pA = p.malloc() 只会分配内存而不会调用构造函数，如果要调用构造函数应该使用 A * const t = p.construct();

❑ singleton_pool（#include <boost/pool/singleton_pool.hpp>）

singleton_pool 和 object_pool 一样，不过它可以定义多个 pool 类型的 object，给它们都分配同样大的内存块，另外 singleton_pool 提供静态方法分配内存，且不用定义对象，如下所示：

```
struct PoolTag{};
typedef boost::singleton_pool<PoolTag,sizeof(int)> User_pool;
int * const t = User_pool::malloc();

my_pool::purge_memory(); // 用完后释放内存
```

❑ pool_allocator（#include <boost/pool/pool_alloc.hpp>）

pool_allocator 基于 singleton_pool 实现，提供 allocator，可用于 STL 等：

```
std::vector<int, pool_allocator<int> > v;
v.push_back(13);
boost::singleton_pool<sizeof(int)>::release_memory(); // 显式调用释放内存
```

由此可见，Boost 确实是一个值得称赞、更值得使用的库，它能为我们提供极大的便利。当需要使用内存池技术时，请考虑 Boost。

请记住：

当你需要频繁地分配相同大小的对象，而又苦恼于默认的内存管理函数带来的问题时，内存池技术将是灵丹妙药，它能提高内存操作效率，以及应用程序的鲁棒性。
