第8章　内 联 函 数

C++继承C的一个重要特性是效率。假如 C++的效率显著地比 C低，程序设计者不会使用它。

在C中，保护效率的一个方法是使用宏 (macro)。宏可以不用普通函数调用就使之看起来像函数调用。宏的实现是用预处理器而不是编译器。预处理器直接用宏代码代替宏调用，所以就没有了参数压栈、生成汇编语言的 CALL、返回参数、执行汇编语言的 RETURN的时间花费。所有的工作由预处理器完成，因此，不用花费什么就具有了程序调用的便利和可读性。

C++中，使用预处理器宏存在两个问题。第一个问题在 C中也存在：宏看起来像一个函数调用，但并不总是这样。这就隐藏了难以发现的错误。第二个问题是 C++特有的：预处理器不容许存取私有(private)数据。这意味着预处理器宏在用作成员函数时变得非常无用。

为了既保持预处理器宏的效率又增加安全性，而且还能像一般成员函数一样可以在类里访问自如，C++用了内联函数(inline function)。本章我们将研究C++预处理器宏存在的问题、C++中如何用内联函数解决这些问题以及使用内联函数的方针。

8.1　预处理器的缺陷

预处理器宏存在的关键问题是我们可能认为预处理器的行为和编译器的行为一样。当然，有意使宏在外观上和行为上与函数调用一样，因此容易被混淆。当微妙的差异出现时，问题就出现了。

考虑下面这个简单例子：

```
#define f (x) (x+1)
```

现在假如有一个像下面的f的调用

```
f (1)
```

预处理器展开它，出现下面不希望的情况：

```
(x)  (x+1) (1)
```

出现这个问题是因为在宏定义中 f和括号之间存在空格缝隙。当定义中的这个空格取消后，实际上调用宏时可以有空格空隙。像下面的调用：

```
f   (1)
```

依然可以正确地展开为：

```
(1 + 1)
```

上面的例子虽然微不足道但问题非常明显。在宏调用中使用表达式作为参数时，问题就出现了。

存在两个问题。第一个问题是表达式在宏内展开，所以它们的优先级不同于我们所期望的优先级。例如：

```
#define floor(x,b) x>=b?0:1
```

现在假如对参数使用表达式

```
if(floor(a&0x0f,0x07)) // ...
```

宏将展开成：

```
if(a&0x0f>=0x07?0:1)
```

因为&的优先级比>=的低，所以宏的展开结果将会使我们惊讶。一旦发现这个问题，可以通过在宏定义内使用括弧来解决。上面的定义可改写如下：

```
#define floor(x,b) ((x)>=(b)?0:1)
```

发现问题可能很难，我们可能一直认为宏的行为是正确的。在前面没有加括号的版本的例子中，大多数表达式将正确工作，因为 >=的优先级比像+、/、--甚至位移动操作符的优先级都低。因此，很容易想到它对于所有的表达式都正确，包括那些位逻辑操作符。

前面的问题可以通过谨慎地编程来解决：在宏中将所有的内容都用括号括起来。第二个问题则更加微妙。不像普通函数，每次在宏中使用一个参数，都对这个参数求值。只要宏仅用普通变量调用，这个求值就开始了。但假如参数求值有副作用，那么结果可能出乎预料，并肯定不能模仿函数行为。

例如，下面这个宏决定它的参数是否在一定范围：

```
#define band(x) (((x)>5 && (x)<10) ? (x) : 0)
```

只要使用一个“普通”参数，宏和真的函数工作得非常相像。但只要我们松懈并开始相信它是一个真的函数时，问题就开始出现了。

```
//: MACRO.CPP -- Side effects with macros
#include <fstream.h>
#define band(x) (((x)>5 && (x)<10) ? (x) : 0)

main() {
  ofstream out("macro.out");
  for(int i = 4; i < 11; i++) {
    int a = i;
    out << "a = " << a << endl << '\t';
    out << "band(++a)=" << band(++a) << endl;
    out << "\t a = " << a << endl;
  }
}
```

下面是这个程序的输出，它完全不是我们想从真正的函数期望得到的结果：

```
a = 4
       band(++a)=0
        a = 5
a = 5
       band(++a)=8
        a = 8
a = 6
       band(++a)=9
        a = 9
a = 7
       band(++a)=10
```

```
        a = 10
a = 8
      band(++a)=0
        a = 10
a = 9
      band(++a)=0
        a = 11
a = 10
      band(++a)=0
        a = 12
```

当a等于4时，测试了条件表达式第一部分，但它不满足条件，而表达式只求值一次，所以宏调用的副作用是a等于5，这是在相同的情况下普通函数调用得到的结果。但当数字在范围之内时，两个表达式都测试，产生两次自增操作。产生这个结果是由于再次对参数操作。一旦数字出了范围，两个条件仍然测试，所以也产生两次自增操作。根据参数不同产生的副作用也不同。

很清楚，这不是我们想从看起来像函数调用的宏中所希望的。在这种情况下，明显有效的解决方法是设计真正的函数。当然，如果多次调用函数将会增加额外的开销并可能降低效率。不幸的是，问题可能并不总是如此明显。我们可能不知不觉地得到一个包含混合函数和宏在一起的库函数，所以像这样的问题可能隐藏了一些难以发现的缺陷。例如，在 STDIO.H 中的putc( )宏可能对它的第二个参数求值两次。这在标准 C中作了详细说明。宏toupper( )不谨慎地执行也会对第二个参数求值超过两次。如在使用 toupper(*p++) [1] 时就会产生不希望的结果。

宏和访问

当然，对于C需要对预处理器宏谨慎地编码和使用。即使不是因为宏不是成员函数所需要的范围概念这一原因，我们也会在C++中避免使用它所带来的麻烦。预处理器简单地执行原文替代，所以不可能用下面这样或近似的形式写：

```
class X {
  int i;
public:
#define val (X::i)  //Error
```

另外，这里没有指明我们正在涉及哪个对象。在宏里简直没有办法表示类的范围。没有能取代预处理器宏的方法，程序设计者出于效率考虑，不得不让一些数据成员成为 public类型，这样就会暴露内部的实现并妨碍在这个实现中的改变。

8.2 内联函数

在解决C++中宏存取私有的类成员的问题过程中，所有和预处理器宏有关的问题也随着消失了。这是通过使宏被编译器控制来实现的。在 C++中，宏的概念是作为内联函数来实现的，而内联函数无论在任何意义上都是真正的函数。唯一不同之处是内联函数在适当时像宏一样展开，所以函数调用的开销被取消。因此，应该永远不使用宏，只使用内联函数。

[1] 在Andraw Koenig所著的书《C的陷阱和缺陷》（Addision-Wesley,1989）中将更详细地阐述。

任何在类中定义的函数自动地成为内联函数，但也可以使用 inline关键字放在类外定义的函数前面使之成为内联函数。但为了使之有效，必须使函数体和声明结合在一起，否则，编译器将它作为普通函数对待。因此

```
inline int PlusOne(int x);
```

没有任何效果，仅仅只是声明函数（这不一定能够在稍后某个时候得到一个内联定义）。成功的方法如下：

```
inline int PlusOne(int x) { return  ++x ;}
```

注意，编译器将检查函数参数列表使用是否正确，并返回值（进行必要的转换）。这些事情是预处理器无法完成的。假如对于上面的内联函数，我们写成一个预处理器宏的话，将有不想要的副作用。

一般应该把内联定义放在头文件里。当编译器看到这个定义时，它把函数类型（函数名 + 返回值）和函数体放到符号表里。当使用函数时，编译器检查以确保调用是正确的且返回值被正确使用，然后将函数调用替换为函数体，因而消除了开销。内联代码的确占用空间，但假如函数较小，这实际上比为了一个普通函数调用而产生的代码（参数压栈和执行 CALL）占用的空间还少。

在头文件里，内联函数默认为内部连接——即它是 static，并且只能在它被包含的编译单元看到。因而，只要它们不在相同的编译单元中声明，在内联函数和全局函数之间用同样的名字也不会在连接时产生冲突。

8.2.1 类内部的内联函数

为了定义内联函数，通常必须在函数定义前面放一个 inline关键字。但这在类内部定义内联函数时并不是必须的。任何在类内部定义的函数自动地为内联函数。如下例：

```
//: INLINE.CPP -- Inlines inside classes
#include <iostream.h>

class point {
  int i, j, k;
public:
  point() { i = j = k = 0; }
  point(int I, int J, int K) {
    i = I;
    j = J;
    k = K;
  }
  void print(const char* msg = "") const {
    if(*msg) cout << msg << endl;
    cout << "i = " << i << ", "
         << "j = " << j << ", "
         << "k = " << k << endl;
  }
};

main() {
```

```
  point p, q(1,2,3);
  p.print("value of p");
  q.print("value of q");
}
```

当然，因为类内部的内联函数节省了在外部定义成员函数的额外步骤，所以我们一定想在类声明内每一处都使用内联函数。但应记住，内联的目的是减少函数调用的开销。假如函数较大，那么花费在函数体内的时间相对于进出函数的时间的比例就会较大，所以收获会较小。而且内联一个大函数将会使该函数所有被调用的地方都做代码复制，结果代码膨胀而在速度方面获得的好处却很少或者没有。

8.2.2 存取函数

在类中内联函数的最重要的用处之一是用于一种叫存取函数的函数。这是一个小函数，它容许读或修改对象状态——即一个或几个内部变量。类内存取函数使用内联方式重要的原因在下面的例子中可以看到。

```
//: ACCESS.CPP -- Inline access functions

class access {
  int i;
public:
  int read() const { return i; }
  void set(int I) { i = I; }
};

main() {
  access A;
  A.set(100);
  int x = A.read();
}
```

这里，在类的设计者控制下，将类里面状态变量设计为私有 (private)，类的使用者就永远不会直接和它们发生联系了。对私有 (private)数据成员的所有存取只可以通过成员函数接口进行。而且，这种存取是相当有效的。例如对于函数 read( )。若没用内联函数，对 read( )调用产生的代码将包括对this压栈和执行汇编语言CALL。对于大多数机器，产生的代码将比内联函数产生的代码大一些，执行的时间肯定要长一些。

不用内联函数，考虑效率的类设计者将忍不住简单地使 i为公共(public)成员，从而通过让用户直接存取i 而节约开销。从设计的角度看，这是很不好的。因为 i将成为公共界面的一部分，所以意味着类设计者决不能修改它。我们将和称为 i的一个int类型变量打交道。这是一个问题，因为我们可能在稍后觉得用一个 float变量比用一个int 变量代表状态信息更有用一些，但因为 int i是公共接口的一部分，所以我们不能改变它。另一方面，假如我们总是使用成员函数读和修改一个对象的状态信息，那么就可以满意地修改对象内部一些描述（应该永远打消在编码和测试之前能使我们的设计完善的念头）。

• 存取器(accessors)和修改器(mutators)

一些人进一步把存取函数的概念分成存取器（从一个对象读状态信息）和修改器（修改状

态信息)。而且，可以用重载函数对存取器和修改器提供相同名字的函数，如何调用函数决定了我们是读还是修改状态信息。

```
//: RECTANGL.CPP -- Accessors & mutators

class rectangle {
  int Width, Height;
public:
  rectangle(int W = 0, int H = 0)
    : Width(W), Height(H) {}
  int width() const { return Width; } // Read
  void width(int W) { Width = W; } // Set
  int height() const { return Height; } // Read
  void height(int H) { Height = H; } // Set
};

main() {
  rectangle R(19, 47);
  // Change width & height:
  R.height(2 * R.width());
  R.width(2 * R.height());
}
```

构造函数使用构造函数初始表达式表（在第 7章中作了简介，在13中章将详细介绍）来初始化Width和Height值（对于内部数据类型使用伪编译器调用形式)。

当然，存取器和修改器对于一个内部变量不必只是简单的传递途径。有时，它们可以执行一些计算。下面的例子使用标准的C库函数中的时间函数来生成简单的Time类：

```
//: CPPTIME.H -- A simple time class
#ifndef CPPTIME_H_
#define CPPTIME_H_
#include <time.h>
#include <string.h>

class Time {
  time_t T;
  tm local;
  char Ascii[26];
unsigned char lflag, aflag;
void updateLocal() {
  if(!lflag) {
    local = *localtime(&T);
    lflag++;
  }
}
void updateAscii() {
  if(!aflag) {
```

```
      updateLocal();
      strcpy(Ascii, asctime(&local));
      aflag++;
    }
  }
public:
  Time() { mark(); }
  void mark() {
    lflag = aflag = 0;
    time(&T);
  }
  const char* ascii() {
    updateAscii();
    return Ascii;
  }
  // Difference in seconds:
  int delta(Time* dt) const {
    return difftime(T, dt->T);
  }
  int DaylightSavings() {
    updateLocal();
    return local.tm_isdst;
  }
  int DayOfYear() { // Since January 1
    updateLocal();
    return local.tm_yday;
  }
  int DayOfWeek() { // Since Sunday
  updateLocal();
  return local.tm_wday;
}
int Since1900() { // Years since 1900
  updateLocal();
  return local.tm_year;
}
int Month() { // Since January
  updateLocal();
  return local.tm_mon;
}
int DayOfMonth() {
  updateLocal();
  return local.tm_mday;
}
int Hour() { // Since midight, 24-hour clock
  updateLocal();
  return local.tm_hour;
```

```
  }
  int Minute() {
    updateLocal();
    return local.tm_min;
  }
  int Second() {
    updateLocal();
    return local.tm_sec;
  }
};
#endif // CPPTIME_H_
```

标准C库函数对于时间有多种表示，它们都是类Time的一部分。但全部更新它们是没有必要的，所以time_t T被用作基本的表示法，tm local 和ASCII字符表示法Ascii都有一个标记来显示它们是否已被更新为当前的时间time_t。两个私有函数updateLocal( )和 updateAscii( )检查标记，并有条件地执行更新。

构造函数调用mark( )函数时（用户也可以调用它，强迫对象表示当前时间）也就清除了两个标记，这时当地时间和ASCII表示法是无效的。函数ascii( )调用updateAscii( )， 因为函数ascii( )使用静态数据，假如它被调用，则这个静态数据被重写，所以updateAscaii( )把标准C库函数的结果拷贝到内部缓冲器里。返回值就是内部缓冲器的地址。

所有以DaylightSaving( )开始的函数都使用函数updateLocal( )，这就使得复合的内联函数变得相当大。这似乎不划算，尤其是考虑到可能不经常调用这些函数。但这不意味着所有的函数都应该用非内联函数。假如让updateLocal( )作为一个内联函数，它的代码将被复制在所有的非内联函数里，也能节省额外的开销。

下面是一个小的测试程序：

```
//: CPPTIME.CPP -- Testing a simple time class
#include "..\7\cpptime.h"
#include <iostream.h>

main() {
  Time start;
  for(int i = 1; i < 1000; i++) {
    cout << i << ' ';
    if(i%10 == 0) cout << endl;
  }
  Time end;
  cout << endl;
  cout << "start = " << start.ascii();
  cout << "end = " << end.ascii();
  cout << "delta = " << end.delta(&start);
}
```

在这个例子里，一个Time对象被创建，然后执行一些时延动作，接着创建第2个Time对象来标记结束时间。这些用于显示开始时间、结束时间和消耗的时间。

8.3 内联函数和编译器

为了理解内联何时有效，应该先理解编译器遇到一个内联函数时将做什么。对于任何函数，编译器在它的符号表里放入函数类型（即包括名字和参数类型的函数原型及函数的返回类型）。另外，编译器看到内联函数和内联函数的分析没有错误时，函数的代码也被放入符号表。代码是以源程序形式存放还是以编译过的汇编指令形式存放取决于编译器。

调用一个内联函数时，编译器首先确保调用正确，即所有的参数类型必须是正确类型或编译器必须能够将类型转换为正确类型，并且返回值在目标表达式里应该是正确类型或可改变为正确类型。当然，编译器对任何类型函数都是这样做的，这与预处理器显著不同，因为预处理器不能检查类型和进行转换。

假如所有的函数类型信息符合调用的上下文的话，内联函数代码就会直接替换函数调用，消除了调用的开销。假如内联函数也是成员函数，对象的地址（this)就会被放入合适的地方，这当然也是预处理器不能执行的。

8.3.1 局限性

这儿有两种编译器不能处理内联的情况。在这些情况下，它就像对非内联函数一样，通过定义内联函数和为函数建立存贮空间，简单地将其转换为函数的普通形式。假如它必须在多编译单元里做这些（通常将产生一个多定义错误），连接器就会被告知忽略多重定义。

假如函数太复杂，编译器将不能执行内联。这取决于特定编译器，但大多数编译器这时都会放弃内联方式，因为这时内联将可能不为我们提供任何效率。一般地，任何种类的循环都被认为太复杂而不扩展为内联函数。循环在函数里可能比调用要花费更多的时间。假如函数仅有一条简单语句，编译器可能没有任何内联的麻烦，但假如有许多语句，调用函数的开销将比执行函数体的开销少多了。记住，每次调用一个大的内联函数，整个函数体就被插入在函数调用的地方，所以没有任何引人注目的执行上的改进就使代码膨胀。本书的一些例子可能超过了一定的合理内联尺寸。

假如我们要显式或隐含地取函数地址，编译器也不能执行内联。因为这时编译器必须为函数代码分配内存从而为我们产生一个函数的地址。但当地址不需要时，编译器仍可能内联代码。

我们必须理解内联仅是编译器的一个建议，编译器不强迫内联任何代码。一个好的编译器将会内联小的、简单的函数，同时明智地忽略那些太复杂的内联。这将给我们想要的结果——具有宏效率的函数调用。

8.3.2 赋值顺序

假如我们想象编译器对执行内联做了些什么时，我们可能糊里糊涂地认为存在着比事实上更多的限制。特别是，假如一个内联函数对于一个还没有在类里声明的函数进行向前引用，编译器就可能不能处理它。

```
//: EVORDER.CPP -- Inline evaluation order

class forward {
  int i;
public:
  forward() : i(0) {}
```

```
  // Call to undeclared function:
  int f() const { return g() + 1; }
  int g() const { return i; }
};

main() {
  forward F;
  F.f();
}
```

虽然函数g( )还没有定义，但在函数f( )里对函数g( )进行了调用。这是可行的，因为语言定义规定非内联函数直到类声明结束才赋值。

当然，函数g( )也调用函数f( )，我们将得到一组递归调用，这些递归对于编译器进行内联是过于复杂了。（应该在函数f( )或g( )里也执行一些测试来强迫它们之一“停止”，否则递归将是无穷的）。

8.3.3 在构造函数和析构函数里隐藏行为

构造函数和析构函数是两个使我们易于认为内联比它实际上更有效的函数。构造函数和析构函数都可能隐藏行为，因为类可以包含子对象，子对象的构造函数和析构函数必须被调用。这些子对象可能是成员对象，或可能由于继承（继承还没有介绍）而存在。下面是一个有成员对象的例子。

```
//: HIDDEN.CPP -- Hidden activites in inlines
#include <iostream.h>

class member {
  int i, j, k;
public:
  member(int x = 0) { i = j = k = x; }
  ~member() { cout << "~member" << endl; }
};

class withMembers {
  member Q, R, S; // Have constructors
  int i;
public:
  withMembers(int I) : i(I) {} // Trivial?
  ~withMembers() {
    cout << "~withMembers" << endl;
  }
};

main() {
  withMembers WM(1);
}
```

在类withMembers里，内联的构造函数和析构函数看起来似乎很直接和简单，但其实很复杂。成员对象Q、P和S的构造函数和析构函数被自动调用，这些构造函数和析构函数也是内联的，所以它们和普通的成员函数的差别是显著的。这并不是意味着应该使构造函数和析构函数定义为非内联的。一般说来，快速地写代码来建立一个程序的初始“轮廓”时，使用内联函数经常是便利的。但假如要考虑效率，内联是值得注意的一个问题。

8.4 减少混乱

在本书里，类里放入内联定义的简单性、精练性是非常有用的，因为这样更容易放在一页或一屏中，看起来更方便一些。但Dan Saks指出，在一个真正的工程里，这将造成类接口混乱，因此使类难以使用。他用拉丁文 *in situ* 来表示定义在类里的成员函数(在适当的位置上)，并主张所有的定义都放在类外面以保持接口清楚。他认为这并不妨碍最优化。假如想优化，那么使用关键字inline。使用这个方法，前面（8.2.2节）RECTANGL.CPP例子修改如下：

```
//: NOINSITU.CPP -- Removing in situ functions

class rectangle {
  int Width, Height;
public:
  rectangle(int W = 0, int H = 0);
  int width() const; // Read
  void width(int W); // Set
  int height() const; // Read
  void height(int H); // Set
};

inline rectangle::rectangle(int W, int H)
  : Width(W), Height(H) {
}

inline int rectangle::width() const {
  return Width;
}

inline void rectangle::width(int W) {
  Width = W;
}

inline int rectangle::height() const {
  return Height;
}

inline void rectangle::height(int H) {
  Height = H;
}
```

```
main() {
  rectangle R(19, 47);
  // Transpose width & height:
  R.height(R.width());
  R.width(R.height());
}
```

现在假如想比较一下内联函数与非内联函数的效果，可以简单地移去关键字 inline。（内联函数通常应该放在头文件里，但非内联函数必须放在它们自己的编译单元里。）假如想把函数放入文件，只用简单的剪切和粘贴操作。*In situ*函数需要更多的操作，且更可能出错。这个方法的另外一个争论是我们可能总是对于函数定义使用一致的格式化类型，有些并没有总是在*in situ*函数中出现。

8.5 预处理器的特点

前面我说过，我们几乎总是希望使用内联函数代替预处理器宏。然而当在标准 C预处理器（通过继承也是C++预处理器）里使用3个特别的特征时却是例外：字符串定义、字符串串联和标志粘贴。字符串定义的完成是用 #指示，它容许设一个标识符并把它转化为字符串，然而字符串串联发生在当两个相邻的字符串没有分隔符时，在这种情况下字符串组合在一起。在写调试代码时，这两个特征是非常有效的。

```
#define DEBUG(X)  cout<<#X " = " << X << endl
```

上面的这个定义可以打印任何变量的值。我们也可以得到一个跟踪信息，在此信息里打印出它们执行的语句。

```
#define TRACE(S) cout << #S << endl; S
```

#S定义了要输出的语句。第2个S重申了语句，所以这个语句被执行。当然，这可能会产生问题，尤其是在一行for循环中。

```
for (int i = 0 ; i < 100 ; i++ )
  TRACE(f(i)) ;
```

因为在TRACE( )宏里实际上有两个语句，所以一行 for循环只执行第一个。解决方法是在宏中用逗号代替分号。

标志粘贴

标志粘贴在写代码时是非常有用的。它让我们设两个标识符并把它们粘贴在一起自动产生一个新的标识符。例如：

```
#define FIELD(A) char* A##_string; int A##_size
class record {
  FIELD(one);
  FIELD(two);
  FIELD(three);
  // ...
};
```

每次调用FIELD( )宏，将产生一个保存字符串的标识符和另一个保存字符串长度的标识符。

它不仅易读而且消除了编码出错，使维护更容易。但注意宏的名字中使用大写字母。这是对我们非常有帮助的习惯，因为它告诉读者这是宏而不是函数。所以假如存在问题，它可以作为一个提示。

8.6 改进的错误检查

为本书其余部分改进错误检查是很方便的。用内联函数可以简单地包括一个文件而不用担心连接什么。到目前为止，assert( )宏已用于“错误检查”，但它真正用处是调试并终将被能够在运行时提供更多有用信息的东西代替。何况异常处理程序（在17章介绍）已提供了更多的处理这些错误的有效的方法。

这是预处理器仍然有用的另一个例子，因为_FILE_和_LINE_指示仅和预处理器一起起作用并用在assert( )宏里。假如assert( )宏在一个错误函数里被调用，它仅打印出错函数的行号和文件名字而不是调用错误函数。这儿显示了使用宏联接（许多是 assert( )方法）函数的方法，紧接着调用assert( )（程序调试成功后这由一个#define NDEBUG消除）。

下面的头文件将放在书的根目录中，所以它可以从所有的章节里得到。“Allege”是assert的同义词。

```
//: ALLEGE.H -- Error checking
#ifndef ALLEGE_H_
#define ALLEGE_H_
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

inline void
allege_error(int val, const char * msg){
  if(!val) {
    fprintf(stderr, "error: %s\n", msg);
#ifdef NDEBUG
    exit(1);
#endif
  }
}

#define allege(expr, msg) \
{  allege_error((expr) ? 1 : 0, msg); \
  assert(expr); }

#define allegemem(expr) \
  allege(expr, "out of memory")

#define allegefile(expr) \
  allege(expr, "could not open file")
#endif // ALLEGE_H_
```

函数allege_error( )有两个参数：一个是整型表达式的值，另一个是这个值为 false时需打印

的消息。函数fprintf( )代替iostreams是因为在只有少量错误的情况下，它工作得更好。假如这不是为调试建立的，exit(1)被调用以终止程序。

allege( )宏使用三重if-then-else强迫计算表达式expr求值。在宏里调用了allege_error( )，接着是assert( )，所以我们能在调试时获得 assert( )的好处——因为有些环境紧密地把调试器和assert( )结合在一起。

allegefile( )宏和allegemen( )宏分别是allege( )宏用于检查文件和内存的专用版本。这个代码提供了出错报告的必要的最少信息，但我们可以在这个框架基础上增加它。

下面是测试ALLEGE.H简单例子。

```
//: ERRTEST.CPP -- Testing the allege() macro
// #define NDEBUG // turn off asserts
#include "..\allege.h"
#include <fstream.h>

main() {
  int i = 1;
  allege(i, "value must be nonzero");
  void* m = malloc(100);
  allegemem(m);
  ifstream nofile("nofile.xxx");
  allegefile(nofile);
}
```

去掉下面这行的注释符后，我们就知道这个程序是如何变为成品的：

```
//#define NDEBUG  // turn off asserts
```

对于本书其余部分，将一律用allege( )宏代替assert( )，只有个别只须在调试时检查而运行时不需的情况才用assert( )。

8.7 小结

能够隐藏类下面的实现是关键的，因为在以后我们有可能想修改那个实现。我们可能为了效率这样做，或因为对问题有了更好的理解，或因为有些新类变得可用而想在实现里使用这些新类。任何危害实现隐蔽性的东西都会减少语言的灵活性。这样，内联函数就显得非常重要，因为它实际上消除了预处理器宏和伴随它们的问题。通过用内联函数方式，成员函数可以和预处理器宏一样有效。

当然，内联函数也许会在类定义里被多次使用。因为它更简单，所以程序设计者都会这样做。但这不是大问题，因为以后期待程序规模减少时，可以将函数移出内联而不影响它们的功能。开发指南应该是“首先是使它起作用，然后优化它。”

8.8 练习

1. 将第7章练习2例子增加一个内联构造函数和一个称为Print( )的内联成员函数，这个函数用于打印所有数组的值。

2. 对于第3章的NESTFRND.CPP例子，用内联函数代替所有的成员函数，使它们成为非

*in situ*内联函数。同时再把initialize( )函数改成构造函数。

3. 使用第6章NL.CPP，在它自己的头文件里，将nl转变为内联函数。

4. 创建一个类A，具有能自我宣布的缺省构造函数。再写一个新类B，将A的一个对象作为B的成员，并为类B写一个内联构造函数。创建一个B对象的数组并看看发生了什么事。

5. 从练习4里创建大量的对象并使用Time类来计算非内联构造函数和内联构造函数之间的时间差别（假如我们有剖析器，也试着使用它。）