第二个例子，假设需要一个数据类型Register，它在Pentium上是32位，在UltraSPARC上是64位。这可以由图13-6b中的条件代码来处理（假设编译器产生32位的int和64位的long）。一旦做出这样的定义（可能是在别的什么地方的头文件中），程序员就可以只需声明变量为Register类型并且确信它们将具有正确的长度。

当然，头文件config.h必须正确地定义。对于Pentium处理器，它大概是这样的：

```
#define CPU PENTIUM
#define WORD_LENGTH 32
```

为了编译针对UltraSPARC的系统，应该使用不同的config.h，其中具有针对UltraSPARC的正确取值，它或许是这样的：

```
#define CPU ULTRASPARC
#define WORD_LENGTH 64
```

一些读者可能奇怪为什么CPU和WORD_LENGTH用不同的宏来处理。我们可以很容易地用针对CPU的测试而将Register的定义用括号括起，对于Pentium将其设置为32位，对于UltraSPARC将其设置为64位。然而，这并不是一个好主意。考虑一下以后当我们将系统移植到64位Intel Itanium处理器时会发生什么事情。我们可能不得不为了Itanium而在图13-6b中添加第三个条件。通过像上面那样定义宏，我们要做的全部事情是在config.h文件中为Itanium处理器包含如下的代码行：

```
#define WORD_LENGTH 64
```

这个例子例证了前面讨论过的正交性原则。那些依赖CPU的细节应该基于CPU宏而条件编译，而那些依赖字长的细节则应该使用WORD_LENGTH宏。类似的考虑对于许多其他参数也是适用的。

2. 间接

人们不时地说在计算机科学中没有什么问题不能通过另一个层次间接得到解决。虽然有些夸大其词，但是其中的确存在一定程度的真实性。让我们考虑一些例子。在基于Pentium的系统上，当一个键被按下时，硬件将生成一个中断并且将键的编号而不是ASCII字符编码送到一个设备寄存器中。此外，当此键后来被释放时，第二个中断生成，同样伴随一个键编号。间接为操作系统使用键编号作为索引检索一张表格以获取ASCII字符提供了可能，这使得处理世界上不同国家使用的许多键盘十分容易。获得按下与释放两个信息使得将任何键作为换档键成为可能，因为操作系统知道键按下与释放的准确序列。

间接还被用在输出上。程序可以写ASCII字符到屏幕上，但是这些字符被解释为针对当前输出字体的一张表格的索引。表项包含字符的位图。这一间接使得将字符与字体相分离成为可能。

间接的另一个例子是UNIX中主设备号的使用。在内核内部，有一张表格以块设备的主设备号作为索引，还有另一张表格用于字符设备。当一个进程打开一个特定的文件（例如/dev/hd0）时，系统从i节点提取出类型（块设备或字符设备）和主副设备号，并且检索适当的驱动程序表以找到驱动程序。这一间接使得重新配置系统十分容易，因为程序涉及的是符号化的设备名，而不是实际的驱动程序名。

还有另一个间接的例子出现在消息传递的系统中，该系统命名一个邮箱而不是一个进程作为消息的目的地。通过间接使用邮箱（而不是指定一个进程作为目的地），能够获得相当可观的灵活性（例如，让一位秘书处理她的老板的消息）。

在某种意义上，使用诸如

```
#define PROC_TABLE_SIZE 256
```

的宏也是间接的一种形式，因为程序员无须知道表格实际有多大就可以编写代码。一个良好的习惯是为所有的常量提供符号化的名字（有时-1、0和1除外），并且将它们放在头文件中，同时提供注释解释它们代表什么。

3. 可重用性

在略微不同的上下文中重用相同的代码通常是可行的。这样做是一个很好的想法，因为它减少了二进制代码的大小并且意味着代码只需要调试一次。例如，假设用位图来跟踪磁盘上的空闲块。磁盘块管理可以通过提供管理位图的过程alloc和free得到处理。