

# Fundamental Data Types

第 12 章

## 基本数据类型

[cc2e.com/1278](http://cc2e.com/1278) 内容

- 12.1 数值概论：第 292 页
- 12.2 整数：第 293 页
- 12.3 浮点数：第 295 页
- 12.4 字符和字符串：第 297 页
- 12.5 布尔变量：第 301 页
- 12.6 枚举类型：第 303 页
- 12.7 具名常量：第 307 页
- 12.8 数组：第 310 页
- 12.9 创建你自己的类型（类型别名）：第 311 页

### 相关章节

- 数据命名：第 11 章
- 不常见的数据类型：第 13 章
- 使用变量的一般事项：第 10 章
- 格式化数据声明：第 31.5 节中的“数据声明的布局”
- 注释变量：第 32.5 节中的“注释数据声明”
- 创建类：第 6 章

基本数据类型是构建其他所有数据类型的构造块 (building blocks)。本章包含了使用数（普遍意义上）、整数、浮点数、字符和字符串、布尔变量、枚举类型、具名常量以及数组的一些技巧。本章的最后一节将讲述如何创建自己的数据类型。

本章也涵盖了与基本数据类型有关的问题的基本处理方法。如果你已经了解了关于基本数据类型的内容，就可以跳到本章的最后去查看需要避免的问题的列表，然后去阅读第 13 章有关不常见的数据类型的讨论。

# 12.1 Numbers in General 数值概论

下面一些建议能够使你在使用数的时候少犯错误。

**交叉参考** 关于使用命名常量代替神秘数值的详情，见本章后面的第12.7节“**具名常量**”。

**避免使用“神秘数值（magic number）”** 神秘数值是在程序中出现的、没有经过解释的数值文字量（literal numbers），如100或者47 523。如果你编程用的语言支持具名常量，那么就用它来代替神秘数值。如果你无法使用具名常量，在可行的情况下应该使用全局变量。

避免使用神秘数值会带来以下三点好处：

- 修改会变得更可靠。如果你使用了具名常量，就不会在修改时漏掉多个100中的某一个，或者修改了一个代表其他含义的100。
- 修改会变得更容易。当记录的最大值由100变成200的时候，如果你用了神秘数值，就不得不找出所有的100，然后把它们改成200。如果你用的是100+1或者100-1，你还是要找出所有的101和99来，并把它们改成201和199。如果你用了具名常量，你只需简单地在一处把该常量的定义由100改成200。
- 你的代码变得更可读。当然，对于表达式。

```
for i = 0 to 99 do ...
```

你可以猜测99表示的是数据项的最大数目。但是表达式

```
for i = 0 to MAX_ENTRIES-1 do ...
```

根本就不需要去猜。即使你确信某个数在代码中永远也不会改变，使用具名常量也会有助于提高可读性。

如果需要，可以使用硬编码的0和1 数值0和1用于增量、减量和从数组的第一个元素开始循环。0用于

```
for i = 0 to CONSTANT do ...
```

是可以接受的，把1用在

```
total = total + 1
```

也可以。一条很好的经验法则是，程序主体中仅能出现的文字量就是0和1。任何其他文字量都应该换成更有描述性的表示。

**预防除零（divide-by-zero）错误** 每次使用除法符号的时候（在多数语言里是“/”），都要考虑表达式的分母是否有可能为0。如果这种可能性存在，就应该写代码防止除零错误的发生。

**使类型转换变得明显** 确认当不同数据类型之间的转换发生时，阅读你代码的人会注意到这点。在 C++里你可以使用

```
y = x + (float) i
```

在 Microsoft Visual Basic 里你可以使用

```
y = x + CSng( i )
```

这种实践还能帮助确认有关转换正是你期望发生的——不同的编译器会执行不同的转换，因此，如果不这么做，你就只有碰运气了。

**交叉参考** 第 12.3 节的“**避免等量判断**”提到本例的一种变体。

**避免混合类型的比较** 如果  $x$  是浮点数， $i$  是整数，那么下面的测试

```
if ( i = x ) then ...
```

不能保证可行。在编译器设法弄清了应该用什么类型去进行比较之后，它会把其中一种类型转化为另一种，执行一些四舍五入运算之后才得出结果。要是在这样的情况下你的程序还能跑起来，那就是你的运气了。请自己动手进行类型转换，这样编译器就能比较两个相同类型的数值了，你也会确切地知道它比较的是什么。



KEY POINT

**注意编译器的警告** 当你在同一表达式中使用了多种类型的数值的时候，很多现代的编译器都会通知你。要小心！很多程序员都曾被请去帮助别人解决某个讨厌的错误，结果却发现编译器一直都在对这个错误发出警告。杰出的程序员会修改他们的代码来消除所有的编译器警告。通过编译器警告来发现问题要比你自己找容易得多。

## 12.2 Integers 整数

在用整数的时候，要记住下面的注意事项。

**检查整数除法** 当你使用整数的时候， $7/10$  不等于  $0.7$ 。它总是等于  $0$ ，或者等于负无穷大，或者等于最接近的整数，或者——你应该懂了吧。其结果会随语言的不同而不同。这一说法对中间结果也同样适用。在现实世界中  $10*(7/10) = (10*7) / 10 = 7$ 。但在整数运算的世界里却不同。 $10*(7/10)$  等于  $0$ ，因为整数除法  $(7/10)$  等于  $0$ 。对此问题最简单的补救办法是重新安排表达式的顺序，以最后执行除法： $(10*7)/10$ 。

**检查整数溢出** 在做整数乘法或加法的时候，你要留心可能的最大整数。允许出现的最大无符号整数经常是  $2^{32}-1$ ，有时候是  $2^{16}-1$ ，即  $65\ 535$ 。当你把两个整数相乘，得出的数值大于整数的最大值时，就会出现问题。比如，如果你执行  $250*300$ ，正确的答案是  $75\ 000$ 。但如果最大的整数是  $65\ 535$ ，那么你得到的答案

可能会是 9 464，因为发生了整数溢出 ( $75\ 000 - 65\ 536 = 9\ 464$ )。表 12-1 列出了常用整数类型的取值范围。

表 12-1 不同类型整数的取值范围

整数类型	取值范围
带符号 8-bit 整数	-128 至 127
无符号 8-bit 整数	0 至 255
带符号 16-bit 整数	-32 768 至 32 767
无符号 16-bit 整数	0 至 65 535
带符号 32-bit 整数	-2 147 483 648 至 2 147 483 647
无符号 32-bit 整数	0 至 4 294 967 295
带符号 64-bit 整数	-9 223 372 036 854 775 808 至 9 223 372 036 854 775 807
无符号 64-bit 整数	0 至 18 446 744 073 709 551 615

避免整数溢出的最简单办法是考虑清楚算术表达式中的每个项，设想每项可能达到的最大值。例如，如果在整数表达式  $m = j * k$  中， $j$  可预期的最大值是 200， $k$  可预期的最大值是 25，那么  $m$  可预期的最大值就是  $200 * 25 = 5000$ 。这在 32 位计算机上是没问题的，因为最大的整数是 2 147 483 647。然而，如果  $j$  可预期的最大值是 200 000， $k$  可预期的最大值是 100 000，那么  $m$  可预期的最大值就是  $200\ 000 * 100\ 000 = 20\ 000\ 000\ 000$ 。这时就行不通了，因为 20 000 000 000 要大于 2 147 483 647。在这种情况下，你就必须使用 64 位整数或者浮点数，以容纳  $m$  的预期最大取值。

另外还要考虑程序在未来的扩展。如果  $m$  的取值永远不会超过 5 000，那很好。但如果你预计  $m$  的取值会在几年时间内稳定增长，那么就要把这种情况考虑进来。

**检查中间结果溢出** 你需要关心的不仅是一个算式的最后数值。假设你写有下述代码：

```
Java示例：中间结果溢出
int termA = 1000000;
int termB = 1000000;
int product = termA * termB / 1000000;
System.out.println( "(" + termA + " * " + termB + " ) / 1000000 = " + product );
```

如果你认为  $product$  的赋值结果与  $(1\ 000\ 000 * 1\ 000\ 000) / 1\ 000\ 100$  相等，可能期望所得的结果为 1 000 000。但是这段代码必须先计算出  $1\ 000\ 000 * 1\ 000\ 000$  的中间结果，然后再除以最后面的 1 000 000，而这就意味着它需要一个像 1 000 000 000 000 这么大的数字。你觉得会怎么样？下面就是结果：

```
(1000000 * 1000000) / 1000000 = -727
```

如果整数值最高只能达到 2 147 483 647，那么中间结果对于这一整数数据类型来说实在是太大了。如此一来，本该等于 1 000 000 000 000 的中间结果实际上等于了 -727 379 968。因此，当你用 1 000 000 去除的时候，得到的是 -727，而不是 1 000 000。

你可以用处理整数溢出的相同办法来处理中间结果溢出，换用一种更长的整型或者浮点类型。

## 12.3 Floating-Point Numbers 浮点数



KEY POINT

使用浮点数字时主要考虑的是，很多十进制小数不能够精确地用数字计算机中的 1 和 0 来表示。像  $1/3$  或者  $1/7$  这样的无限循环小数通常只用 7 位或者 15 位精度有效数字表示。在我所用的 Microsoft Visual Basic 版本中， $1/3$  的 32 位浮点数表示形式为 0.33 333 330。它的精确度是小数点后 7 位。这对于大多数用途而言是足够精确的，但是有时它的不精确性也足以给你带来麻烦。

下面是一些在使用浮点数时应该遵循的指导原则。

**交叉参考** 有一些算法方面的书籍描述了解决这些问题的方法，参见第 10.1 节中的“有关数据类型的更多资源”。

解决方案是什么？如果你必须要把一系列差异如此巨大的数相加，那么就先对这些数排序，然后从最小值开始把它们加起来。同样，如果你需要对无穷数列进行求和，那么就从最小的值开始——从本质上来说，是要做逆向的求和运算。这样做并不能消除舍入问题，但是能使这一问题的影响减少到最低限度。很多的算法书都建议采用这种处理方式。

1 等于 2，对充分大的 1 成立。  
—佚名

**避免等量判断** 很多应该相等的浮点数值并不一定相等。这里的根本问题是，用两种不同方法来求同一数值，结果不一定总得到同一个值。举例来说，10 个 0.1 加起来很少会等于 1.0。下面例子显示了应该相等但却不等的两个变量，`nominal` 和 `sum`。

变量 `nominal` 是一个 64 位实数。

`sum` 是  $10 \times 0.1$ ，应当等于 1.0。

这是错误的比较。

### Java示例：对浮点数进行错误的比较

```
double nominal = 1.0;
double sum = 0.0;

for ( int i = 0; i < 10; i++ ) {
    sum += 0.1;
}

if ( nominal == sum ) {
    System.out.println( "Numbers are the same." );
} else {
    System.out.println( "Numbers are different." );
}
```

正如你可能已经猜到的那样，这个程序的输出是

```
Numbers are different.
```

按代码逐行运行，`for` 循环中的 `sum` 值是这样的：

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.799999999999999
0.899999999999999
0.999999999999999
```

因此，应该找一种代替对浮点数字执行等量判断的方案。一种有效的方法是先确定可接受的精确度范围，然后用布尔函数判断数值是否足够接近。通常应该写一个 `Equals()` 函数，如果数值足够接近就返回 `true`，否则就返回 `false`。在 Java 中，这样的函数类似下面这样：

**交叉参考** 这个例子印证了每条规则皆有例外这一哲理。这个实际例子里的变量名包含了数字。在第 11.7 节的“应该避免的名字”里提到过避免在变量名中使用数字这一规则。

#### Java示例：比较浮点数的函数

```
final double ACCEPTABLE_DELTA = 0.00001;
boolean Equals( double Term1, double Term2 ) {
    if ( Math.abs( Term1 - Term2 ) < ACCEPTABLE_DELTA ) {
        return true;
    }
    else {
        return false;
    }
}
```

如果修改了前面对浮点数做出错误比较的例子中的代码，改用上述函数来做比较，那么新的比较就会是：

```
if ( Equals( Nominal, Sum ) ) ...
```

使用了这样的比较方法后，程序的输出就会变成：

```
Numbers are the same.
```

根据程序需求，对 `ACCEPTABLE_DELTA` 的值进行硬编码可能是不合适的。你也许需要根据待比较的两个数的大小算出 `ACCEPTABLE_DELTA`。

**处理舍入误差问题** 由于舍入误差而导致的错误与由于数字之间数量级相差太大而导致的错误并无二致。问题相同，解决的技术也相同。除此之外，下面列出一些专门用于解决舍入问题的常见方案。

- 换用一种精确度更高的变量类型。如果你正在用单精度浮点值，那么就换用双精度浮点值，同理类推。
- 换用二进制编码的十进制（binary coded decimal, BCD）变量。BCD 模式的处理通常更慢一些，并且要占用更多的存储空间，但是它能防止很多舍入错误的发生。当你使用的变量代表的是美元、美分或者其他必须要精确结算的数量的时候，这种方法会特别有用。
- 把浮点变量变成整型变量。这是一种自力更生转到 BCD 变量的方法。你可能必须要用 64 位整数才能获得所需的精度。采用这种方法要求你自己来处理数字的小数部分。假设你原来是用浮点数处理美元，其中美分表示为美元的小数部分。这是一种常用的处理美元和美分的方式。当你转到用整数的时候，就必须用整数来表示美分，用美分的 100 倍来表示美元。换句话说，你把美元乘以 100，并把美分保存到变量值中 0 到 99 的范围内。这样做乍一看有点别扭，但是无论从速度还是精确度的角度来看，它都是一种有效的解决方案。你可以创建一个能够隐藏整数表示并且支持必要数字运算的 DollarsAndCents 类，来简化这些操作。

**交叉参考** 通常，将二进制编码转换为 BCD 对性能产生的影响是很小的。如果你对这一影响仍心存疑虑，可以看看第 25.6 节“代码调整方法总结”。

**检查语言和函数库对特定数据类型的支持** 有些语言，包括 Visual Basic 在内，包含了像 Currency 这样的数据类型，专用于处理对舍入误差敏感的数据。如果你的语言中内置了提供此类功能的数据类型，那么就用它！

## 12.4 Characters and Strings 字符和字符串

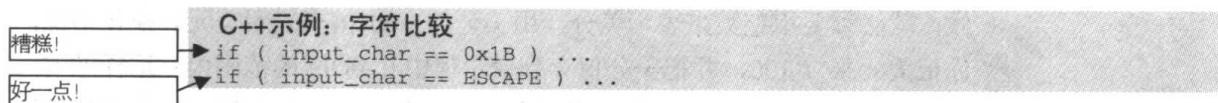
本节给出一些使用字符串的技巧。其中的第一条适用于所有的语言。

**交叉参考** 在第 12.1 节“数值概论”中有关于神秘数值的描述，里面提到的一些事项同样适用于神秘字符以及神秘字符串。

**避免使用神秘字符和神秘字符串** 神秘字符（magic character）是指程序中随处可见的字面形式表示的字符（literal character，例如‘A’），神秘字符串（magic string）是指字面形式表示的字符串（literal string，例如“Gigamatic Accounting Program”）。如果你用的编程语言支持具名常量，则用具名常量来加以取代。否则就用全局变量。避免使用字面形式的字符串的众多原因如下：

- 对于程序的名字、命令名称、报表标题等常常出现的字符串，你有时可能需要修改它们的内容。例如，“Gigamatic Accounting Program”可能会在一个新版本里改为“New and Improved! Gigamatic Accouting Program”。

- 国际市场的重要意义正在日益突现，翻译存放在字符串资源文件中的字符串要比翻译存在于代码中的字符串容易得多。
- 字符串的字面表示形式通常都会占用较多的存储空间。它用于菜单、消息、帮助屏幕、录入表格等。如果字符串的数量太多，就会失控，并引发内存问题。在很多环境中，字符串的存储空间并不是要考虑的因素，但是在嵌入式系统开发以及其他存储空间非常珍贵的应用中，这一点就必须考虑。在那些情况下，如果字符串是相对独立于源代码的，那么针对字符串空间问题的解决方案就会更容易实施。
- 字符和字符串的字面表示形式的含义是模糊的。注释或具名常量能够澄清你的意图。在下例中，`0x1B` 的含义并不清楚。使用 `ESCAPE` 常量使得这一含义变得更加明显了。



**避免 off-by-one 错误** 由于子字符串的下标索引方式几乎与数组相同，因此要避免因为读写操作超出了字符串末尾而导致的 off-by-one（偏差一）错误。

[cc2e.com/1285](http://cc2e.com/1285)

**了解你的语言和开发环境是如何支持 Unicode 的** 在 Java 等语言里，所有的字符串都是 Unicode 的。在 C 和 C++ 等其他的语言里，处理 Unicode 就要用到与之相关的一组函数。为了标准函数库与第三方函数库之间的通信，常常需要在 Unicode 和其他的字符集之间进行转换。如果有些字符串不需要表示成 Unicode（例如，在 C 或 C++ 中），就要尽早决定是否采用 Unicode 字符集。如果你决定要用 Unicode 字符串，就要决定何处以及何时使用它。

**在程序生命期中尽早决定国际化/本地化策略** 与国际化和本地化相关的事项都是很重要的问题。关键的考虑事项包括：决定是否把所有字符串保存在外部资源里，是否为每一种语言创建单独的版本，或者在运行时确定特定的界面语言<sup>1</sup>。

[cc2e.com/1292](http://cc2e.com/1292)

**如果你知道只需要支持一种文字的语言，请考虑使用 ISO 8859 字符集** 对于只需要支持单一文字（例如英语）、无须支持多语言或者某种表意语言（例如汉语）的应用程序，可以使用 ISO 8859 扩展 ASCII 类型标准来很好地替代 Unicode。

<sup>1</sup>译注：指自然语言。

如果你需要支持多种语言, 请使用 Unicode。与 ISO 8859 或者其他标准相比, Unicode 对国际字符集提供了更为全面的支持。

**采用某种一致的字符串类型转换策略** 如果你使用了多种字符串类型, 有一种常用方法能维护各种字符串类型, 那就是在程序中把所有字符串都保存为一种格式, 同时在尽可能靠近输入和输出操作的位置把字符串转换为其他格式。

## Strings in C C 语言中的字符串

C++ 的标准模板库 (STL) 中的 `string` 类已经消除了大多数与 C 中字符串相关的传统问题。下面为那些直接处理 C 字符串的程序员提供一些避免常见错误的方法。

**注意字符串指针和字符数组之间的差异** 与字符串指针 (string pointer) 和字符数组 (character array) 相关的问题来源于 C 处理字符串的方式。请从以下两方面留心二者之间的差异。

- 警惕任何包含字符串和等号的表达式。C 中的字符串操作差不多都是通过 `strcmp()`、`strcpy()`、`strlen()` 及其他相关的子程序完成的。等号几乎总意味着某种指针错误。在 C 里面, 赋值并不把字符串字面量 (string literal) 拷贝给字符串变量。假设你写有下面这行语句

```
StringPtr = "Some Text String";
```

在这种情况下, "Some Text String" 是一个指向字面量字符串的指针, 因此这个赋值的结果只是让 `StringPtr` 指针指向该字面字符串。这个赋值并没有把字符串内容拷贝给 `StringPtr`。

- 通过命名规则区分变量是字符数组还是字符串指针。一种常见的规则是用 `ps` 前缀来标识字符串指针, 用 `ach` 前缀来标识字符数组。尽管同时含有 `ps` 和 `ach` 前缀的表达式不一定总是错的, 但你还是应该对它们持怀疑态度。

**把 C-style 字符串的长度声明为 `CONSTANT + 1`** 在 C 和 C++ 中, 与 C-style 字符串相关的 off-by-one 错误很常见, 因为很容易忘记长度为 `n` 的字符串需要 `n + 1` 字节的存储空间, 从而忘记为空结束符 (位于字符串的最后取值为 0 的字节) 预留空间。避免这类错误的一种有效方法是用具名常量来声明所有字符串。这种方法的关键之处在于你每次都用同样的方式使用该具名常量。把字符串的长度声明为 `CONSTANT + 1`, 然后在余下的代码里用 `CONSTANT` 来指代字符串的长度。下面举一个例子:

**C示例：好的字符串声明**

```

/* Declare the string to have length of "constant+1".
   Every other place in the program, "constant" rather
   than "constant+1" is used. */
→char name[ NAME_LENGTH + 1 ] = { 0 }; /* string of length NAME_LENGTH */

...
/* Example 1: Set the string to all 'A's using the constant,
   NAME_LENGTH, as the number of 'A's that can be copied.
   Note that NAME_LENGTH rather than NAME_LENGTH + 1 is used. */
→for ( i = 0; i < NAME_LENGTH; i++ )
    name[ i ] = 'A';
...
/* Example 2: Copy another string into the first string using
   the constant as the maximum length that can be copied. */
→strncpy( name, some_other_name, NAME_LENGTH );

```

这里声明的字符串长度为  
NAME\_LENGTH+1

在这里，对字符串的操作用到了  
NAME\_LENGTH

这里也用到了。

如果你没有采用上面的规则来处理这一问题，你就会有时把字符串的长度声明为 NAME\_LENGTH，然后在操作中使用 NAME\_LENGTH - 1；而有时你会把字符串长度声明为 NAME\_LENGTH + 1，然后在操作中使用 NAME\_LENGTH。每次使用字符串的时候你都不得不去想自己是如何声明它的。

当每次都用相同的方式来使用字符串的时候，你就不需要去记住每个字符串是怎么处理的，从而避免由于忘记某字符串的处理细节而导致的错误。使用规则有助于减少脑力消耗以及编程失误。

**交叉参考 第 10.3 节“变量初始化原则”有关于数据初始化的详细介绍。** **用 null 初始化字符串以避免没有终端的字符串** C 通过查找空结束符，即字符串末尾取值为 0 的字节，来判断字符串的末尾。不管你认为字符串有多长，只要 C 没有找到空结束符，它就认为字符串还没有结束。如果你忘记在字符串的最后放置一个空值，字符串操作的结果可能就会与你预想的不一样。

你可以用两种方法来避免无终端的字符串。首先，在声明字符数组的时候把它初始化为 0：

**C示例：声明字符数组的好做法**

```
char EventName[ MAX_NAME_LENGTH + 1 ] = { 0 };
```

其次，在你动态分配字符串的时候，使用 `calloc()` 而不是 `malloc()` 来把它初始化为 0。`calloc()` 会负责分配内存，并把它初始化为 0。`malloc()` 只分配内存，并不执行初始化，因此，当你使用由 `malloc()` 分配的内存的时候就要小心了。

**交叉参考** 本章后面的第 12.8 节“数组”详细描述了数组。

**用字符数组取代 C 中的指针** 如果内存不是限制性的因素——通常都不是——那么就把你所有的字符串变量都声明为字符数组。这样做有助于避免指针错误，并且在出错的时候，编译器会给你更多的警告。

**用 strncpy()取代 strcpy()以避免无终端的字符串** C 中的字符串子程序既有安全版本，也有危险版本。较危险的子程序，如 `strcpy()` 和 `strcmp()`，会一直运行下去，直到它们遇到了一个空结束符为止。安全版本即 `strncpy()` 和 `strncmp()`，会接受一个表示最大长度的参数，因此一旦处理到此参数长度位置，即使字符串会一直延续下去，你的函数调用也会及时返回。<sup>2</sup>

## 12.5 Boolean Variables 布尔变量

要把逻辑变量或者布尔变量用错是非常困难的，而更仔细地运用它会让你的程序变得更清晰。

**交叉参考** 第 32 章“自说明代码”详细介绍了如何使用注释对程序做进一步说明。

**用布尔变量对程序加以文档说明** 不同于仅仅判断一个布尔表达式，你可以把这种表达式的结果赋给一个变量，从而使得这一判断的含义变得明显。例如，在下面的代码片断中，`if` 检查的对象到底是工作完成、错误条件还是其他什么，情况很不明确：

**交叉参考** 第 19.1 节“简化复杂的表达式”里有一个借助布尔函数来对程序提供说明的例子。

### Java示例：目的不明确的布尔判断

```
if ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex ) ||  
    ( elementIndex == lastElementIndex )  
) {  
    ...  
}
```

在下面这段代码中，布尔变量的使用使得 `if` 的判断对象明确多了：

### Java示例：目的明确的布尔判断

```
finished = ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex ) );  
repeatedEntry = ( elementIndex == lastElementIndex );  
if ( finished || repeatedEntry ) {  
    ...  
}
```

**用布尔变量来简化复杂的判断** 常有这样的情况，在需要编写一段复杂的判断时，你要尝试好几次才能成功。在你事后想要修改这一判断的时候，首先弄清楚这段判断在做什么就已经很困难了。逻辑变量可以简化这种判断。在前述示例中，程序事实上需要判断两个条件：子程序是否已经结束，以及子程序是否在重复的记录上工作。通过创建 `finished` 和 `repeatedEntry` 两个布尔变量，`if` 的判断得到了简化：读起来很容易，更不容易出错，修改起来也更加方便了。

<sup>2</sup>译注：更安全的做法是使用 `strlcpy()` 或 `strcpy_s()`。

下面再举一个复杂判断的例子：



#### Visual Basic示例：复杂的判断

```
If ( ( document.AtEndOfStream() ) And ( Not inputError ) ) And _  
    ( ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES ) ) And _  
    ( Not ErrorProcessing() ) Then  
    ' do something or other  
    ...  
End If
```

本例中的判断相当复杂，但是这种情况并不罕见。它给读者很重的思维负担。我猜你甚至不会去试着理解 if 判断的含义，而是看着它说，“要是需要的话，我会以后再去弄清楚。”请注意这种想法，因为这就是别人阅读你所写的含有类似判断语句代码时的真切反应。

下面是重写的代码，增加了布尔变量以简化判断：

这是简化之后的判断。

#### Visual Basic示例：简化后的判断

```
allDataRead = ( document.AtEndOfStream() ) And ( Not inputError )  
legalLineCount = ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES )  
If ( allDataRead ) And ( legalLineCount ) And ( Not ErrorProcessing() ) Then  
    ' do something or other  
    ...  
End If
```

第二个版本更简单些。我想你阅读这个 if 判断里的布尔表达式不会有任何困难。

**如果需要的话，创建你自己的布尔类型** 有些语言，比如 C++、Java 和 Visual Basic，含有预定义的布尔类型。其他语言，比如 C，却没有。在 C 这样的语言中，你可以定义你自己的布尔类型。在 C 中你可能会这样做：

#### C示例：用typedef定义BOOLEAN类型

```
typedef int BOOLEAN;
```

或者你也可以这么做，其额外好处是同时定义出了 true 和 false：

#### C示例：用枚举定义Boolean类型

```
enum Boolean {  
    True=1,  
    False=(!True)  
};
```

把变量声明为 BOOLEAN 而非 int，可以让其用途更为明显，并且使你的程序不言自明。

## 12.6 Enumerated Types 枚举类型

枚举类型是一种允许用英语来描述某一类对象中每一个成员的数据类型。C++ 和 Visual Basic 都提供了枚举类型，通常用在你知道变量的所有可能取值，并且希望把它们用单词表达出来的时候。下面举一些 Visual Basic 中的枚举类型示例：

### Visual Basic示例：枚举类型

```
Public Enum Color
    Color_Red
    Color_Green
    Color_Blue
End Enum

Public Enum Country
    Country_China
    Country_England
    Country_France
    Country_Germany
    Country_India
    Country_Japan
    Country_Usa
End Enum

Public Enum Output
    Output_Screen
    Output_Printer
    Output_File
End Enum
```

枚举类型是老式说明方法的强有力替代者，使用老式说明方法的时候你需要明确地说，“1 代表红色，2 代表绿色，3 代表蓝色……”。下面给出一些如何使用枚举类型的指导原则。

### 用枚举类型来提高可读性 与下面这个语句相比

```
if chosenColor = 1
```

你可以通过下面这样的语句来提高可读性

```
if chosenColor = Color_Red
```

每当你看到字面形式数字的时候，就应该问问自己，把它换成枚举类型是不是更合理。

枚举类型特别适用于定义子程序参数。有谁知道下面函数调用里的参数代表什么？

**C++示例：函数调用——用枚举会更好**

```
int result = RetrievePayrollData( data, true, false, false, true );
```

与之相反，下面函数调用里的参数更容易理解：

**C++示例：函数调用——使用枚举提高可读性**

```
int result = RetrievePayrollData(
    data,
    EmploymentStatus_CurrentEmployee,
    PayrollType_Salaried,
    SavingsPlan_NoDeduction,
    MedicalCoverage_IncludeDependents
);
```

**用枚举类型来提高可靠性** 对于少数语言而言（尤其是 Ada），枚举类型会使编译器执行比整数和常量更为彻底的类型检查。如果采用具名常量，编译器将无法知道仅有 Color\_Red, Color\_Green 和 Color\_Blue 是合法的值。编译器不会反对像 color = Country\_England 或者 country = Output\_Printer 这样的语句。但如果你用了枚举类型，把一个变量声明为 Color，编译器就会只允许把该变量赋值为 Color\_Red, Color\_Green 或 Color\_Blue。

**用枚举类型来简化修改** 枚举类型使得你的代码更容易修改。如果你在“1 代表红色，2 代表绿色，3 代表蓝色”方案中发现了一处缺陷，你就必须从头到尾检查代码，并且修改所有的 1、2、3 等。如果用的是枚举类型，你就可以继续向列表增加元素，只要把它们加入类型定义后重新编译就可以了。

**将枚举类型作为布尔变量的替换方案** 布尔变量往往无法充分表达它所需要表达的含义。举例而言，假设你有一个子程序在成功地完成任务之后返回 true，否则返回 false。后来你可能发现事实上有两种 false。第一种表示任务失败了，并且其影响只局部于子程序自身；第二种表示任务失败了，而且产生了一个致命错误，需要把它传播到程序的其余部分。在这种情况下，一个包含 Status\_Success、Status\_Warning 和 Status\_FatalError 值的枚举类型，就比一个包含 true 和 false 的布尔类型更有用。如果成功和失败的具体类型有所增加，对其进行扩展以区分这些情况也是非常容易的。

**检查非法数值** 在 if 或者 case 语句中测试枚举类型时，务必记得检查非法值。在 case 语句中用 else 子句捕捉非法值：

**Visual Basic示例：检查枚举类型数据中的无效值**

```

Select Case screenColor
    Case Color_Red
        ...
    Case Color_Blue
        ...
    Case Color_Green
        ...
Case Else
    DisplayInternalError( False, "Internal Error 752: Invalid color." )
End Select

```

这是对无效值的  
判断。

· 定义出枚举的第一项和最后一项，以便用于循环边界 把枚举的第一个和最后一个元素定义成 Color\_First, Color\_Last, Country\_First, Country\_Last 等，使你更方便地写出能遍历所有枚举元素的循环来。你可以用明确的数值来定义该枚举类型，如下所示：

**Visual Basic示例：设置枚举类型数据第一项和最后一项**

```

Public Enum Country
    Country_First = 0
    Country_China = 0
    Country_England = 1
    Country_France = 2
    Country_Germany = 3
    Country_India = 4
    Country_Japan = 5
    Country_Usa = 6
    Country_Last = 6
End Enum

```

现在就可以把 Country\_First 和 Country\_Last 用做循环边界了：

**Visual Basic示例：遍历枚举类型数据元素**

```

' compute currency conversions from US currency to target currency
Dim usaCurrencyConversionRate( Country_Last ) As Single
Dim iCountry As Country
For iCountry = Country_First To Country_Last
    usaCurrencyConversionRate( iCountry ) = ConversionRate( Country_Usa, iCountry )
Next

```

· 把枚举类型的第一个元素留做非法值 在你声明枚举类型的时候，把第一个值保留为非法值。很多编译器会把枚举类型中的第一个元素赋值为 0。把映射到 0 的那个元素声明为无效会有助于捕捉那些没有合理初始化的变量，因为这些变量值更有可能为 0，而不是其他的非法值。

下面就是采用了这种方法后的 Country 声明：

**Visual Basic示例：将枚举中第一个元素声明为无效值**

```
Public Enum Country
    Country_InvalidFirst = 0
    Country_First = 1
    Country_China = 1
    Country_England = 2
    Country_France = 3
    Country_Germany = 4
    Country_India = 5
    Country_Japan = 6
    Country_Usa = 7
    Country_Last = 7
End Enum
```

**明确定义项目代码编写标准中第一个和最后一个元素的使用规则，并且在使用时保持一致** 在枚举中使用 InvalidFirst, First 和 Last 元素，能使数组声明和循环更具有可读性。但是这样做也可能会造成混乱，究竟枚举中的合法项是从 0 开始还是从 1 开始的？枚举中的第一个和最后一个元素合法吗？如果使用这种技术，项目的编码标准中就应该要求在所有的枚举中都统一使用 InvalidFirst、First、Last，以减少出错。

**警惕给枚举元素明确赋值而带来的失误** 有些语言允许对枚举里面的各项元素明确地赋值，如下面这个 C++例子所示：

**对枚举元素直接赋值的C++范例**

```
enum Color {
    Color_InvalidFirst = 0,
    Color_First = 1,
    Color_Red = 1,
    Color_Green = 2,
    Color_Blue = 4,
    Color_Black = 8,
    Color_Last = 8
};
```

在这个例子中，如果你把一个循环的下标声明为 Color 类型，并且尝试去遍历所有的 Color，那么你在遍历 1, 2, 4, 8 这些合法数值的同时，也会遍历 3, 5, 6, 7 这些非法数值。

## If Your Language Doesn't Have Enumerated Types 如果你的语言里没有枚举类型

如果你的语言里没有枚举类型，那么可以用全局变量或者类来模拟它。例如，可以在 Java 中使用下面这些声明：

**交叉参考** 在我写这一章的时候，Java 还不支持枚举类型。等你阅读本章的时候，它或许能够支持了。这正是我在第 4.3 节“你在技术浪潮中的位置”中写到的“翻滚的技术潮流”的一个极好实例。

```
Java示例：模拟枚举类型
// set up Country enumerated type
class Country {
    private Country() {}
    public static final Country China = new Country();
    public static final Country England = new Country();
    public static final Country France = new Country();
    public static final Country Germany = new Country();
    public static final Country India = new Country();
    public static final Country Japan = new Country();
}

// set up Output enumerated type
class Output {
    private Output() {}
    public static final Output Screen = new Output();
    public static final Output Printer = new Output();
    public static final Output File = new Output();
}
```

这些枚举类型会增强你的程序的可读性，因为你可以用 `Country.England` 和 `Output.Screen` 等公用类成员来代替具名常量。这种特殊的创建枚举类型的方法还是类型安全 (type safe) 的；因为每种类型都声明为类，因此编译器会检查非法的赋值，比如 `Output output = Country.England` (Bloch 2001)。

在不支持类的语言中，你也可以通过对全局变量的规范应用来模拟枚举类型中的每一个元素，从而获得同样的基本效果。

## 12.7 Named Constants 具名常量

具名常量很像变量，一旦赋值以后就不能再修改了。具名常量允许你用一个名字而不是数字——比如说 `MAXIMUM_EMPLOYEES` 而不是 1 000——来表示固定的量，比如员工人数的最大值。

使用具名常量是一种将程序“参数化”的方法——把程序中可能变化的一个方面写为一个参数，当需要对其修改时，只改动一处就可以了，而不必在程序中到处改动。如果你曾经声明过一个你认为大小肯定够用的数组，后来却因为容量不够而用光了存储空间，你就会赞美具名常量的作用了。一旦数组的大小变了，

你只需要修改声明该数组时所用的那个常量的定义。这种“单点控制 (single-point control)”对让软件真正地“软”了许多——用起来和改起来都很方便。

**在数据声明中使用具名常量** 在需要定义所用数据的大小的数据声明和其他语句里，使用具名常量可以提高程序的可读性和可维护性。在下例中，就用 LOCAL\_NUMBER\_LENGTH 来描述员工电话号码的长度，而不用数字 7。

**Visual Basic示例：在数据声明中使用具名常量**

```

LOCAL_NUM-  
BER_LENGTH在  
此处声明为常量。  
在这里使用。  
这里也用到了。
Const AREA_CODE_LENGTH = 3
Const LOCAL_NUMBER_LENGTH = 7
...
Type PHONE_NUMBER
    areaCode( AREA_CODE_LENGTH ) As String
    localNumber( LOCAL_NUMBER_LENGTH ) As String
End Type
...
'make sure all characters in phone number are digits
For iDigit = 1 To LOCAL_NUMBER_LENGTH
    If ( phoneNumber.localNumber( iDigit ) < "0" ) Or _
        ( "9" < phoneNumber.localNumber( iDigit ) ) Then
        ' do some error processing
    ...

```

这个例子很简单，但是你可以设想这样一个程序，其中很多地方都用到有关于员工电话号码的信息。

在你创建这个程序的时候，所有的员工都在同一个国家里，所以你只需要 7 位数字就能容纳他们的电话号码。随着规模的扩张，公司在很多国家都建立了分支机构，你就会用到更长的电话号码。如果你将号码长度参数化，那么就可以只修改一处：即 LOCAL\_NUMBER\_LENGTH 具名常量的定义。

**交叉参考** 单点控制的意义在《Software Conflict》(Glass 1991) 第 57 至 60 页有详细叙述。

正如你所料到的那样，使用具名常量非常有助于程序的维护。作为一项一般性的原则，任何一种能够对可能发生改变的事物进行集中控制的技术，都是减少维护工作量的好技术 (Glass 1991)。

**避免使用文字量，即使是“安全”的** 在下面的循环里，你认为 12 代表着什么含义？

**Visual Basic示例：含义模糊的代码**

```

For i = 1 To 12
    profit( i ) = revenue( i ) - expense( i )
Next

```

根据这段代码的特殊本质，看上去它可能是在遍历一年里的 12 个月。不过你能确定吗？你敢用你收藏的 Monty Python 全集来打赌吗？<sup>3</sup>

在这种情况下，你是不大需要用具名常量来支持将来的灵活性的：一年中月份的数量在任何时间都不太可能改变。但是如果代码的写法会让人对它的作用产生任何一丝疑虑，就应该用命名良好的常量来明确它，如下所示：

**Visual Basic示例：含义清晰的代码**

```
For i = 1 To NUM_MONTHS_IN_YEAR
    profit(i) = revenue(i) - expense(i)
Next
```

这样好多了，但是，为了完成这个例子，循环下标的名字也应该能反映出更多的信息来：

**Visual Basic示例：含义更加清晰的代码**

```
For month = 1 To NUM_MONTHS_IN_YEAR
    profit(month) = revenue(month) - expense(month)
Next
```

这看上去已经非常不错了，但是我们还可以用枚举类型来让它更上一层楼：

**Visual Basic示例：含义一目了然的代码**

```
For month = Month_January To Month_December
    profit(month) = revenue(month) - expense(month)
Next
```

对于最后的这个示例，没有人会对循环的用途产生怀疑了。即使你认为使用文字量（literal）是安全的，也应当转去使用具名常量。请成为从代码中剔除文字量的狂热者吧。用一款文本编辑器来寻找代码里的 2、3、4、5、6、7、8 和 9，以确认你没有由于不小心而使用了它们。

**交叉参考** 前面第12.6节中的“如果你的语言里没有枚举类型”有关于如何模拟枚举类型的详细介绍。

**用具有适当作用域的变量或类来模拟具名常量** 如果你的语言不支持具名常量，你可以自行创建解决方案。通过使用与前面模拟枚举类型 Java 示例中建议的相似的方法，你同样可以获得具名常量的优点。需要遵循的典型的作用域原则是：优先选用局部作用域，其次是类作用域，再次是全局作用域。

**统一地使用具名常量** 如果需要表示的是同一个实体，在一处使用具名常量，而在另一处使用数字符号是非常危险的。有些编程实践是在自找麻烦；就像是在拨打 800 免费热线，并要求把错误送上门来。如果某个具名常量的值需要修改，

<sup>3</sup>译注：Monty Python 为 20 世纪 60 年代英国经典电视连续剧，Python 语言由此得名。

你就会去修改它，然后自信已经做了全部所需的改动。这样就会忽略掉那些硬编码的数字符号，从而给你的程序带来难于发现的问题，而解决这些问题可要比抓起电话大声求助难得多。

## 12.8 Arrays 数组



KEY POINT

数组是最简单和最常用的结构化数据类型。在有些语言里，数组是唯一的结构化数据类型。一个数组中含有一组类型完全相同，并且可以用数组下标来直接访问的条目。下面就如何使用数组给出一些建议。

**确认所有的数组下标都没有超出数组的边界** 在任何情况下，与数组有关的所有问题都源于一个事实：数组里的元素可以随机访问。最常见的问题就是程序试图用超出数组边界的下标去访问数组元素。在有些语言里，这种情况会产生一个错误；在其他语言里，这样做会产生一个奇怪的不可预料的结果。

**考虑用容器来取代数组，或者将数组作为顺序化结构来处理** 计算机科学界的一些最聪明的人士建议永远不要随机地访问数组，只能顺序地访问（Mills and Linger 1986）。他们的论点是，在数组里随机访问就像在程序里面随便使用的 `goto` 语句一样：这种访问很容易变得难于管理且容易出错，要证明其是否正确也很困难。他们建议使用集合、栈和队列等按顺序存取元素的数据结构来取代数组。



HARD DATA

在一项小型试验里，Mills 和 Linger 发现按照这种方法所创建的设计中只需要更少的变量和变量引用。相对而言，这样做设计的工作效率较高，能产生高度可靠的软件。

在你习惯性地选用数组之前，考虑能否用其他可以顺序访问数据的容器类作为替换方案——如集合、栈、队列等。

**交叉参考** 使用数组和使用循环所遇到的问题有相似之处，二者也是紧密联系在一起的。在第 16 章“控制循环”有关循环的详细介绍。

**检查数组的边界点** 正如考虑循环结构的边界是非常有用的一样，你可以通过检查数组的边界点来捕获很多错误。问问自己，代码有没有正确地访问数组的第一个元素？还是错误地去访问了第一个元素之前或者之后的那个元素？而后一个元素呢？代码会导致 `off-by-one` 的错误吗？最后，问问你自己代码有没有正确地访问数组中间的元素。

**如果数组是多维的，确认下标的使用顺序是正确的** 很容易把 `Array[j][i]` 写成 `Array[i][j]`，所以请花些时间检查下标的顺序是否正确。与其用 `i` 和 `j` 这类不明不白的东西，不如去考虑更有意义的名字。

**提防下标串话** 如果你在使用嵌套循环，那么会很容易把 `Array[i]` 写成了 `Array[j]`。调换循环下标称为“下标串话（cross-talk）”。请检查这种问题。更好的做法是使用比 `i` 和 `j` 更有意义的下标名，从而在一开始就降低下标串话错误的发生几率。

**在 C 中结合 ARRAY\_LENGTH() 宏来使用数组** 通过定义类似于下面的 `ARRAY_LENGTH()` 宏，你可以更加灵活地使用数组：

**C示例：定义ARRAY\_LENGTH()宏**

```
#define ARRAY_LENGTH( x ) (sizeof(x)/sizeof(x[0]))
```

在你操作数组的时候，用 `ARRAY_LENGTH()` 宏取代具名常量来表示数组大小的上限。如下例所示：

**C示例：借助ARRAY\_LENGTH()宏对数组进行操作**

```
ConsistencyRatios[] =  
{ 0.0, 0.0, 0.58, 0.90, 1.12,  
1.24, 1.32, 1.41, 1.45, 1.49,  
1.51, 1.48, 1.56, 1.57, 1.59 };  
...  
for ( ratioIdx = 0; ratioIdx < ARRAY_LENGTH( ConsistencyRatios ); ratioIdx++ );  
...
```

这里使用了宏。

这种技术对于例子中出现的这种一维数组特别有用。如果你增加或者减少了数组中的条目，你不需要记着去改变用于描述数组大小的具名常量。当然，这种技术也同样适用于多维数组，并且，如果用了这种方法，你就没有必要总为定义数组而多创建一个具名常量。

## 12.9 Creating Your Own Types (Type Aliasing) 创建你自己的类型（类型别名）



KEY POINT

程序员自定义的数据类型是语言所能赋予你的一种最强有力的、最有助于澄清你对程序的理解的功能之一。它保护你的程序免受预料之外更改之困扰，并使得程序更容易阅读——所有这一切都不需要你去设计、构造或者测试新的类。如果你在使用 C、C++ 或者另外一种能够支持用户自定义类型的语言，就请好好利用这些自定义类型！

**交叉参考** 很多情况下，创建一个类会比创建单个数据类型好得多。在第 6 章“可以工作的类”里面有详细的介绍。

为了感受创建自定义类型的威力，假设你正在写一个程序，把 `x`、`y`、`z` 坐标系中的坐标值转化为纬度、经度和海拔高度。你觉得可能需要用双精度浮点值，但除非能对此完全肯定，否则你宁愿用单精度浮点值写程序。你可以使用 C 或 C++ 中的 `typedef` 语句或者其他语言中的相关语句来为坐标创建一个新的特殊类型。

下面是你用 C++ 创建该类型定义的代码：

**C++示例：创建一个数据类型**

```
typedef float Coordinate; // for coordinate variables
```

该类型定义声明了一个新的类型，Coordinate，其功能与 float 类型完全相同。在使用这一新类型时，你就像使用 float 等预定义类型一样用它来声明变量。下面就是一例：

**C++示例：使用前面创建的数据类型**

```
Routine1( ... ) {
    Coordinate latitude; // latitude in degrees
    Coordinate longitude; // longitude in degrees
    Coordinate elevation; // elevation in meters from earth center
    ...
}

Routine2( ... ) {
    Coordinate x; // x coordinate in meters
    Coordinate y; // y coordinate in meters
    Coordinate z; // z coordinate in meters
    ...
}
```

在这段代码里，变量 latitude（纬度），longitude（经度），elevation（海拔），x，y 和 z 都声明为 Coordinate 类型。

现在假设程序发生了变化，你最终发现自己需要使用双精度变量来表示坐标。由于你已经专门为坐标数据定义了一种类型，因此唯一需要修改的就是类型的定义。而且只需要在一处做修改：typedef 语句里面。下面是修改后的类型定义：

**C++示例：改变后的类型定义**

最初的float改为  
double。

```
→typedef double Coordinate; // for coordinate variables
```

下面再举一例——这回用的是 Pascal。假设你在开发一套薪资系统，其中员工姓名的最大长度不超过 30 个字符。你的用户已经说过没有任何人的姓名超出 30 个字符。你会在你的程序里到处硬编码 30 这一数字吗？如果你这么做，那么你对你用户的信任要远远超过我对我用户的信任！更好的做法是为员工姓名定义一种类型：

**Pascal示例：为雇员姓名创建数据类型**

Type

```
employeeName = array[ 1..30 ] of char;
```

一旦涉及到使用字符串或者数组，最明智的做法通常是定义出一个能够表明该字符串或数组长度的具名常量，然后在类型定义中使用该具名常量。程序中有很多地方会用到该具名常量——这里只是第一个位置。代码如下：

**Pascal示例：为雇员姓名创建数据类型——更好的做法**

```
Const
  NAME_LENGTH = 30;
  ...
Type
  employeeName = array[ 1..NAME_LENGTH ] of char;
```

这里定义了具名常量。  
这里使用了前面定义的具名常量。

一个更强大的例子是把创建自己的类型和信息隐藏这两种理念结合起来。在一些情况下，你想要隐藏的信息就是该数据的类型信息。

上面 C++ 的坐标示例只是部分地实现了信息隐藏。如果你总是使用 Coordinate 而非 float 或者 double，你就有效地隐藏了数据的类型。在 C++ 里，这差不多就是语言能够为你隐藏的全部信息。除此之外，你或者你代码的后续使用者还必须遵守“不去查看 Coordinate 的定义”的纪律。C++ 为你提供了象征性的、而不是字面的信息隐藏的能力。

其他的语言，例如 Ada，走得更进一步，支持字面的信息隐藏(literal information hiding)。下面是一个声明了 Coordinate 代码段的 Ada 包 (package) 的样子：

**Ada示例：将类型细节隐藏到包内部**

```
package Transformation is
  type Coordinate is private;
  ...
  
```

这条语句声明  
Coordinate是这个包的私用成员。

下面是另一个使用了 Coordinate 的包的例子：

**Ada示例：使用另一个包内的类型**

```
with Transformation;
...
procedure Routine1(...) ...
  latitude: Coordinate;
  longitude: Coordinate;
begin
  -- statements using latitude and longitude
  ...
end Routine1;
```

请注意，Coordinate 类型在包规格中声明为私用的。这意味着程序中唯一了解 Coordinate 类型定义的部分是 Transformation 包的私用部分。在一个团队开发的环境里，你可以只分发包规格，从而使开发其他包的程序员不能查看 Coordinate 的底层类型。该信息被字面地隐藏了。像 C++ 这样要求你通过头文件来分发 Coordinate 定义的语言里较难实现真正的信息隐藏。

这些例子阐明了多项创建你自己类型的原因。

- **易于修改** 创建一个新类型并不费事，而且它为你带来了很多灵活度。
- **避免过多的信息分发** 采用硬编码而非集中在一处管理数据的方式会导致数据类型的细节散布于程序内部。这是第 6.2 节讨论的集中化的信息隐藏原则时的一个例子。
- **增加可靠性** 在 Ada 中，你可以定义 type Age is range 0..99 这样的类型。编译器接着就会生成运行时检查，以验证所有 Age 类型变量的取值都处于 0 到 99 的范围内。
- **弥补语言的不足** 如果你的语言不具有你所需要的预定义类型，你可以自己来创建它。例如，C 没有布尔或者逻辑类型。这种不足很容易通过自己创建该类型来予以弥补：

```
typedef int Boolean;
```

## Why Are the Examples of Creating Your Own Types in Pascal and Ada

### 为什么创建自己的类型的示例是用 Pascal 和 Ada 写的

Pascal 和 Ada 已经在走向灭亡，而且一般来说，取代它们的语言都更好用。然而，就简单类型定义而言，我认为 C++、Java 和 Visual Basic 在这方面改善不大。Ada 声明如下：

```
currentTemperature: INTEGER range 0..212;
```

包含了下述语句所不具备的重要语义信息：

```
int temperature;
```

再进一步，像下面这样的类型声明：

```
type Temperature is range 0..212;
...
currentTemperature: Temperature;
```

使编译器能保证只把 `currentTemperature` 赋给其他 `Temperature` 类型的变量，这样只需要很少的额外代码就能为程序提供更多的安全边界。

当然，程序员可以创建一个 `Temperature` 类，去推行 Ada 语言里自动推行的同样的语义。但从写一行代码创建的一个简单数据类型，到创建一个类，是很大的一步。在许多情况下，程序员可能愿意创建这个简单类型，但却不愿意向前走一步，付诸更多努力去创建一个类。

## Guidelines for Creating Your Own Types 创建自定义数据类型的指导原则

**交叉参考** 在任何情况下，都应该考虑用类是否会比用简单数据类型更好，相关内容在第 6 章“可以工作的类”中有详细的介绍。

请在创建自己的“用户自定义”类型时考虑下述原则。

**给所创建的类型取功能导向的名字** 避免使用那些代表了类型底层计算机数据类的类型名。应该用能代表该新类型所表现的现实世界问题的类型名。前面例子中的定义就为坐标和人员姓名创建了命名良好的类型——它们代表了现实世界中的事物。与之相似，你可以为货币、支付代码、年龄等——现实世界问题的方方面面——创建类型。

要提防创建了引用预定义类型的类型名。像 `BigInteger` 或 `LongString` 这样的类型名所反映的是计算机数据，而非现实世界问题。创建自定义类型的最大优点，就在于它提供了介于你的程序和实现语言之间的一层绝缘层。引用了底层编程语言类型的类型名就是在该绝缘层上戳了一个洞。它不会比使用一种预定义类型给你带来更多好处。另一方面，以现实问题为导向的名字也使自定义类型容易修改，其作用不言自明。

**避免使用预定义类型** 如果类型有一丝变化的可能，就应避免在除 `typedef` 或类型定义之外的任何位置使用预定义的类型。创建功能导向的新类型很容易，但是修改那些使用了硬编码的类型的程序里的数据却很难。更何况使用功能导向的类型声明，实际上部分地解释了那些通过它们声明的变量。像 `Coordinate x` 这样的声明要比 `float x` 这样的声明告诉你更多有关 `x` 的信息。请尽可能多地使用自己创建的类型。

**不要重定义一个预定义的类型** 改变一个标准类型的定义会引起混淆。例如，如果你的语言有一个预定义的类型 `Integer`，那么就不要创建名为 `Integer` 的自定义类型。代码的读者可能会忘记你已经重新定义了该类型，并认为他们所看到的 `Integer` 就是他们习惯看到的那个 `Integer`。

**定义替代类型以便于移植** 与不要修改标准类型的建议相反，你可能需要为标准类型定义替代类型，以便让变量在不同的硬件平台上正确地代表相同的实体。例如，你可以定义一个 `INT32` 类型，用它来代替 `int`，或者定义 `LONG64`

类型来代替 long。最初，这样的两个类型之间唯一的区别就是它们名字的大小写不同。但是当你把程序移植到一个新的硬件平台之上的时候，你就可以重新定义大写的那个类型版本，以便它们能够与原始硬件的数据类型相匹配。

一定不要定义容易被错认为是预定义类型的类型。或许可以定义 INT 而非 INT32，但你最好把自定义类型和语言所提供的类型明显地区分开来。

**考虑创建一个类而不是使用 typedef** 简单的 typedef 对隐藏变量的底层类型信息是大有帮助的。然而，在一些情况下，你可能会需要定义类所能获得的那些额外的灵活度和控制力。详细信息请见第 6 章“可以工作的类”。

### CHECKLIST: Fundamental Data

### 核对表：基本数据类型

[cc2e.com/1206](http://cc2e.com/1206)

**交叉参考** 如果想看针对普遍数据类型而非特定数据类型的核对表，请看第 10 章 257 页的“使用变量的一般事项”。如果想看如何为变量命名的注意事项核对表，请看第 11 章 288 页的“变量名的力量”。

#### 数值概论

- 代码中避免使用神秘数值吗？
- 代码考虑了除零错误吗？
- 类型转换很明显吗？
- 如果在一条语句中存在两个不同类型的变量，那么这条语句会像你期望的那样求值吗？
- 代码避免了混合类型比较吗？
- 程序编译时没有警告信息吗？

#### 整数

- 使用整数除法的表达式能按预期的那样工作吗？
- 整数表达式避免整数溢出问题吗？

#### 浮点数

- 代码避免了对数量级相差巨大的数字做加减运算吗？
- 代码系统地阻止了舍入错误的发生吗？
- 代码避免对浮点数做等量比较吗？

#### 字符和字符串

- 代码避免使用神秘字符和神秘字符串吗？
- 使用字符串时避免了 off-by-one 错误吗？

- C 代码把字符串指针和字符数组区别对待了吗？
- C 代码遵循了把字符串声明为 `CONSTANT + 1` 长度的规则了吗？
- C 代码在适当的时候用字符数组来代替指针了吗？
- C 代码把字符串初始化为 `NULL` 来避免无终端的字符串了吗？
- C 代码用 `strncpy()` 替代 `strcpy()` 吗？`strncat()` 和 `strcmp()` 呢？

### 布尔变量

- 程序用额外的布尔变量来说明条件判断了吗？
- 程序用额外的布尔变量来简化条件判断了吗？

### 枚举类型

- 程序用枚举类型而非具名常量来提高可读性、可靠性和可修改性吗？
- 当变量的用法不能仅用 `true` 和 `false` 表示的时候，程序用枚举类型来取代布尔变量吗？
- 针对枚举类型的测试检测了非法数值吗？
- 把枚举类型的第一项条目保留为“非法的”了吗？

### 具名常量

- 程序用具名常量而不是神秘数值来声明数据和表示循环界限吗？
- 具名常量的使用一致吗？——没有在有些位置使用具名常量又在其他位置使用文字量？

### 数组

- 所有的数组下标都没有超出数组边界吗？
- 数组引用没有出现 `off-by-one` 错误吗？
- 所有多维数组的下标的顺序都正确吗？
- 在嵌套循环里，把正确的变量用于数组下标来避免循环下标串话了吗？

**创建类型**

- 程序对每一种可能变化的数据分别采用不同的类型吗？
- 类型名是以该类型所表示的现实世界实体为导向，而不是以编程语言类型为导向的吗？
- 类型名的描述性足以强，可以帮助解释数据声明吗？
- 你避免重新定义预定义类型吗？
- 与简单地重定义一个类型相比，你考虑过创建一个新类吗？

**Key Points****要点**

- 使用特定的数据类型就意味着要记住适用于各个类型的很多独立的原则。用本章的核对表来确认你已经对常见问题做了考虑。
- 如果你的语言支持，创建自定义类型会使得你的程序更容易修改，并更具有自描述性。
- 当你用 `typedef` 或者其等价方式创建了一个简单类型的时候，考虑是否更应该创建一个新的类。