


在一台 x86-64/Linux 机器上, double 类型是 8 个字节, 而 int 类型是 4 个字节。在我们的系统中, x 的地址是 0x601020, y 的地址是 0x601024。因此, bar5.c 的第 6 行中的赋值 `x=-0.0` 将用负零的双精度浮点表示覆盖内存中 x 和 y 的位置(foo5.c 中的第 5 行和第 6 行)!

```
linux> gcc -Wall -Og -o foobar5 foo5.c bar5.c
/usr/bin/ld: Warning: alignment 4 of symbol 'x' in /tmp/cc1UfK5g.o
is smaller than 8 in /tmp/ccbTLcb9.o
linux> ./foobar5
x = 0x0 y = 0x80000000
```

这是一个细微而令人讨厌的错误, 尤其是因为它只会触发链接器发出一条警告, 而且通常要在程序执行很久以后才表现出来, 且远离错误发生地。在一个拥有成百上千个模块的大型系统中, 这种类型的错误相当难以修正, 尤其因为许多程序员根本不知道链接器是如何工作的。当你怀疑有此类错误时, 用像 GCC-fno-common 标志这样的选项调用链接器, 这个选项会告诉链接器, 在遇到多重定义的全局符号时, 触发一个错误。或者使用 -Werror 选项, 它会把所有的警告都变为错误。

在 7.5 节中, 我们看到了编译器如何按照一个看似绝对的规则来把符号分配为 COMMON 和 .bss。实际上, 采用这个惯例是由于在某些情况中链接器允许多个模块定义同名的全局符号。当编译器在翻译某个模块时, 遇到一个弱全局符号, 比如说 x, 它并不知道其他模块是否也定义了 x, 如果是, 它无法预测链接器该使用 x 的多重定义中的哪一个。所以编译器把 x 分配成 COMMON, 把决定权留给链接器。另一方面, 如果 x 初始化为 0, 那么它是一个强符号(因此根据规则 2 必须是唯一的), 所以编译器可以很自信地将它分配成 .bss。类似地, 静态符号的构造就必须是唯一的, 所以编译器可以自信地把它们分配成 .data 或 .bss。

 **练习题 7.2** 在此题中, REF(x.i)→DEF(x.k) 表示链接器将把模块 i 中对符号 x 的任意引用与模块 k 中 x 的定义关联起来。对于下面的每个示例, 用这种表示法来说明链接器将如何解析每个模块中对多重定义符号的引用。如果有一个链接时错误(规则 1), 写“错误”。如果链接器从定义中任意选择一个(规则 3), 则写“未知”。

- A.

/* Module 1 */ int main() { } (a) REF(main.1) → DEF(_____._____) (b) REF(main.2) → DEF(_____._____)	/* Module 2 */ int main; int p2() { }
--	---
- B.

/* Module 1 */ void main() { } (a) REF(main.1) → DEF(_____._____) (b) REF(main.2) → DEF(_____._____)	/* Module 2 */ int main = 1; int p2() { }
---	---
- C.

/* Module 1 */ int x; void main() { } (a) REF(main.1) → DEF(_____._____) (b) REF(main.2) → DEF(_____._____)	/* Module 2 */ double x = 1.0; int p2() { }
---	---