(a) REF(x.1) → DEF(_____._____)

(b) REF(x.2) → DEF(_____._____)

7.6.2 与静态库链接

迄今为止，我们都是假设链接器读取一组可重定位目标文件，并把它们链接起来，形成一个输出的可执行文件。实际上，所有的编译系统都提供一种机制，将所有相关的目标模块打包成为一个单独的文件，称为*静态库*(static library)，它可以用做链接器的输入。当链接器构造一个输出的可执行文件时，它只复制静态库里被应用程序引用的目标模块。

为什么系统要支持库的概念呢？以 ISO C99 为例，它定义了一组广泛的标准 I/O、字符串操作和整数数学函数，例如 atoi、printf、scanf、strcpy 和 rand。它们在 libc.a 库中，对每个 C 程序来说都是可用的。ISO C99 还在 libm.a 库中定义了一组广泛的浮点数学函数，例如 sin、cos 和 sqrt。

让我们来看看如果不使用静态库，编译器开发人员会使用什么方法来向用户提供这些函数。一种方法是让编译器辨认出对标准函数的调用，并直接生成相应的代码。Pascal(只提供了一小部分标准函数)采用的就是这种方法，但是这种方法对 C 而言是不合适的，因为 C 标准定义了大量的标准函数。这种方法将给编译器增加显著的复杂性，而且每次添加、删除或修改一个标准函数时，就需要一个新的编译器版本。然而，对于应用程序员而言，这种方法会是非常方便的，因为标准函数将总是可用的。

另一种方法是将所有的标准 C 函数都放在一个单独的可重定位目标模块中(比如说 libc.o 中)应用程序员可以把这个模块链接到他们的可执行文件中：

```
linux> gcc main.c /usr/lib/libc.o
```

这种方法的优点是它将编译器的实现与标准函数的实现分离开来，并且仍然对程序员保持适度的便利。然而，一个很大的缺点是系统中每个可执行文件现在都包含着一份标准函数集合的完全副本，这对磁盘空间是很大的浪费。(在一个典型的系统上，libc.a 大约是 5MB，而 libm.a 大约是 2MB。)更糟的是，每个正在运行的程序都将它自己的这些函数的副本放在内存中，这是对内存的极度浪费。另一个大的缺点是，对任何标准函数的任何改变，无论多么小的改变，都要求库的开发人员重新编译整个源文件，这是一个非常耗时的操作，使得标准函数的开发和维护变得很复杂。

我们可以通过为每个标准函数创建一个独立的可重定位文件，把它们存放在一个为大家都知道的目录中来解决其中的一些问题。然而，这种方法要求应用程序员显式地链接合适的目标模块到它们的可执行文件中，这是一个容易出错而且耗时的过程：

```
linux> gcc main.c /usr/lib/printf.o /usr/lib/scanf.o ...
```

静态库概念被提出来，以解决这些不同方法的缺点。相关的函数可以被编译为独立的目标模块，然后封装成一个单独的静态库文件。然后，应用程序可以通过在命令行上指定单独的文件名字来使用这些在库中定义的函数。比如，使用 C 标准库和数学库中函数的程序可以用形式如下的命令行来编译和链接：

```
linux> gcc main.c /usr/lib/libm.a /usr/lib/libc.a
```

在链接时，链接器将只复制被程序引用的目标模块，这就减少了可执行文件在磁盘和内存中的大小。另一方面，应用程序员只需要包含较少的库文件的名字(实际上，C 编译器驱