你试过这样写 C 程序吗

Functional Programming in C

张泽鹏

2013-8-17

C 语言；结构化编程；面向对象编程；函数式编程

[λ]你试过这样写 C 程序吗

目录

摘要

面向对象风格和函数式编程风格是编写代码的两种风格，面向对象风格早为大众所认知，函数式风格也渐渐受到大家的关注。网上为其布道的文章不少，有人赞扬有人不屑，但鲜有展示一个完整例子的。例如很多人对函数式风格的印象只是“有人说它很好，但不清楚到底好在哪儿，更不知如何在实际的项目中获益”。

本文将采用 C 语言解决一个问题，围绕这个问题不断地变化需求、重构代码，分别展示两种风格如何从不同的侧面提高代码的可维护性。如果你没有耐心读完这篇长文章，可以参见附录 II 直接看代码，但这篇文章会向你解释为什么代码会写成这样，以及写成这样的好处。

注：本文纯属个人观点，如有雷同，非常荣幸！

什么是函数式风格？

面向对象风格大家都耳熟能详，而提到函数式风格，脑海中或多或少会闪过一些耳熟能详的名词：无副作用、无状态、易于并行编程，甚至是 Lisp 那扭曲的前缀表达式。追根溯源，函数式风格源自λ 演算：函数能作为值传递给其他函数或由其他函数返回。其中“函数”是一种抽象的概念，可以理解成代码块，在 C 语言里叫函数或过程，在 Java 中叫成员方法……因此，函数式风格的本质是函数作为“第一等公民”。在我看来，诸如闭包、匿名函数等特性仅是添头，例如 Emacs Lisp 最初不支持闭包，但不影响它是一门支持函数式风格的编程语言。

有些人会把函数式风格与面向对象风格对立起来，但在我看来这两种风格都是为了提高代码的可维护性，可以相辅相成：

- 函数式风格重点是增强类型系统：一些编程语言提供的基础数据类型仅有数值型和字符串型，函数式风格要求函数也是基础数据类型，即代码也是一种数据；
- 面向对象风格侧重代码的组织形式：要求把数据和操作这些数据的函数组织在同一个类中，提高内聚；对象之间通过调用开放的接口通讯，降低耦合。

代码即数据的作用？

使用不支持函数式风格的编程语言开发，将迫使我们永远在语言恰好提供的基础功能上工作。例如迭代只能使用 for、while 等关键字；读写文件每次都要写

fopen、fclose；并行加锁也少不了 lock 和 unlock。面对这些大同小异的冗余代码总会很无奈：如果 XX 语言能提供 XX 特性该多好啊！

代码即数据让这一切成为可能，它允许你自定义控制语句。后文将展示如何自定义控制语句，以及它如何提高代码的可维护性。

为什么选 C 语言？

函数若要作为“第一等公民”，至少需要满足以下四条特权：

1. 可以用变量命名；
2. 可以提供给过程作为参数；
3. 可以由过程作为结果返回；
4. 可以包含在数据结构中。

对照之下会惊讶地发现，C 这门看似离函数式风格最远的编程语言居然也符合上述条件；此外，相比其他对函数式风格支持更好的语言（如 Lisp、Haskell 等），至少 C 的语法不那么古怪；何况熟悉 C 语系（如 Java、C#等）语法的同学也更多，方便大家用自己熟悉的语言实践。

问题描述

作为贯穿全文的主线，这有一个问题需要你开发一个 C 程序来完成任务：有一个存有职员信息（姓名、年龄、工资）的文件“work.txt”，内容如下：

```
William 35 25000
Kishore 41 35000
Wallace 37 20000
Bruce 39 15000
```

1. 要求从文件中读取这些信息，并输出到屏幕上；
2. 为所有工资小于三万的员工涨 3000 元；
3. 在屏幕上输出薪资调整后的结果；
4. 把调整后的结果保存到原始文件。

即运行的结果是屏幕上要有八行输出，“work.txt”的内容将变成：

```
William 35 28000
Kishore 41 35000
Wallace 37 23000
Bruce 39 18000
```

快速实现

这个问题很简单，简单到把所有代码都塞到 main 函数里也不觉得太长：

```
#include <stdio.h>

int main(void) {
    struct {
        char name[8];
        int age;
        int salary;
    } e[4];
    FILE *istream, *ostream;
    int i, length;

    istream = fopen("work.txt", "r");
    for (i = 0; fscanf(istream, "%s%d%d", e[i].name, &e[i].age, &e[i].salary) == 3; i++)
        printf("%s %d %d\n", e[i].name, e[i].age, e[i].salary);
    length = i;
    fclose(istream);

    ostream = fopen("work.txt", "w");
    for (i = 0; i < length; i++) {
        if (e[i].salary < 30000)
            e[i].salary += 3000;
        printf("%s %d %d\n", e[i].name, e[i].age, e[i].salary);
        fprintf(ostream, "%s %d %d\n", e[i].name, e[i].age, e[i].salary);
    }
    fclose(ostream);

    return 0;
}
```

其中第一个循环不断地从 work.txt 中读数据，直到文件末尾，同时把信息输出到屏幕，即实现了需求#1；第二个循环遍历所有数据，为薪资小于三万的职员增加三千元（需求#2），并把调整后的结果输出屏幕（需求#3）和 work.txt 中（需求#4）。

当变化来临时

上面的代码简洁明了，而且运行良好，作为应付无需维护、需求亦不会变化的课后作业绰绰有余。可惜，我们没有活在新闻联播里，需求总在不断地变化，以至于要不停地维护代码。下面从维护的角度罗列几个问题，并尝试重构。

当文件打开失败时

程序发布之后，就面临各种苛刻的运行环境，例如文件 work.txt 可能没有读或写权限。代码的维护者需要通过错误日志里的信息定位出错的位置，但不是所有环境都会提供充足的信息，例如 Linux 下，没有读或写权限都只输出“Segmentation fault”，仅凭这段错误信息无法确定是哪一句 fopen 出错。

当职员信息数量变化时

样例中只有 4 条记录，不意味着真实环境中永远只有 4 条记录，甚至可以认为记录的数目是不确定的。臆断结构体数组的最大长度是 4 或其他数值都是不合适的，需要能自适应不同的数目。

当字段类型变化时

虽然样例中工资都是整数，但真实环境中工资很可能是浮点数。把 int salary 改成 float salary 意味着所有涉及输入输出的地方都要修改：%d 换成%f。

在短短不到 30 行的代码里尚且有 4 处需要修改，换成庞大的项目，维护成本将不可估量。

当字段数目变化时

客户提出职员信息中需新增一列，保存员工入职的年份。这带来的影响和上个问题一样。

当业务逻辑变化时

本例的业务逻辑就是调薪和输出，几乎都集中在了第二个循环体中。如果不断地增加新的业务逻辑，循环体就会爆炸式地增长。而且业务逻辑可能需要相互组合，代码就变得杂乱无章。

面向对象风格

上节提到的变化都很常见，你肯定还能想出更多。它们综合的维护成本已不比完全重写低，即代码应对合理需求变化的能力差，可维护性低。究其原因，是相同或相似的代码散落在多处，因此一个变化就引起多处更改，误改或漏改都在所难免。

回顾前文面向对象风格的宗旨：把数据和操作数据的函数集中在一起，开放操作数据的接口供其它对象或方法调用。这恰好能解决把操作数据的方法散落在各处的问题，下面就用面向对象的思想重构代码。

抽象数据结构

首先需要抽象出要处理的对象类型，此处为结构体命名即可：

```
typedef struct _Employee {
    …
} *Employee;
```

需要注意的是，Employee 是结构体_Employee 的指针，因为操作结构体，使用指针比直接使用对象更频繁。

接着要选择一种数据结构作为容器，由于数据是一组个数不确定的线性结构，单链表正好适合这样的场景：

```
typedef struct _Employee {
    String name;
    int age;
    int salary;
    struct _Employee *next;
} *Employee;
```

开放接口

根据需求，职员对象至少提供从文件中读取信息、输出到屏幕、保存到文件、调整薪资四项功能。其中输出到屏幕和保存到文件可以合并成输出到输出流中，因此它将开放以下四个接口：

1. employee_read：批量从输入流中读取职员信息并返回
2. employee_free：批量释放动态申请的空间
3. employee_print：批量输出职员信息到输出流
4. employee_adjust_salary：遍历职员信息并调整薪资

构造函数

即创建并初始化对象的函数。

```
static Employee employee_read_node(File istream) {
   Employee e = (Employee)calloc(1, sizeof(struct _Employee));
   if (e != NULL && fscanf(istream, "%s%d%d", e->name, &e->age, &e->salary) != 3) {
      employee_free(e);
      e = NULL;
   }
   return e;
}
```

构造函数先通过 calloc 申请了一片内存空间（并自动初始化为 0），再从给定的输入流中读取职员信息来初始化对象，如果输入流中没有更多的数据，就释放空间并返回空指针。

该函数只能构造单个对象，而文件中有一组对象，且需要串联成单链表结构，因此接口 employee_read 的工作就是组织这些对象：

```
Employee employee_read(File istream) {
   Employee e = NULL, head = NULL, tail = NULL;

   while (e = employee_read_node(istream)) {
      if (head != NULL) {
         tail->next = e;
         tail = e;
      } else {
         head = tail = e;
      }
   }

   return head;
}
```

由于 employee_read_node 是一个辅助函数，不是对外开放的接口，所以使用 static 修饰把作用域限定在当前文件。

析构函数

因为对象的空间是动态申请的，需要提供手工释放的析构函数，即 employee_free：

```
void employee_free(Employee e) {
    Employee p;
    while (p = e) {
        e = e->next;
        free(p);
    }
}
```

输出

如果说输入是把字符串反序列化成对象的过程，那输出就是输入的逆运算，即把对象序列化成字符串的过程。因此，输出的要求是格式必须和输入文件保持一致，允许程序多次处理。此处的输出就是遍历整个集合并输出到输出流：

```
void employee_print(File ostream, Employee e) {
    for (; e; e = e->next) {
        fprintf(ostream, "%s %d %d\n", e->name, e->age, e->salary);
    }
}
```

核心业务逻辑：调整薪资

与输出类似，调整薪资也是遍历整个集合，为符合要求的职员调薪：

```
void employee_adjust_salary(Employee e) {
    for (; e; e = e->next) {
        if (e->salary < 30000) {
            e->salary += 3000;
        }
    }
}
```

解决问题

经过以上几个步骤，为职员信息管理这个领域定义了一套方便的接口。此时的 main 函数不用再操心数据具体以什么形式组织、如何获取、如何输出，只需向 Employee 对象发送消息（调用接口）即可完成任务。

```
int main(void) {
    File istream, ostream;
    Employee e = NULL;

    istream = fopen("work.txt", "r");
```

```
    if (istream == NULL) {
        fprintf(stderr, "Cannot open work.txt with r mode.\n");
        exit(EXIT_FAILURE);
    }
    e = employee_read(istream);
    fclose(istream);

    employee_print(stdout, e);

    employee_adjust_salary(e);

    employee_print(stdout, e);

    ostream = fopen("work.txt", "w");
    if (ostream == NULL) {
        fprintf(stderr, "Cannot open work.txt with w mode.\n");
        exit(EXIT_FAILURE);
    }
    employee_print(ostream, e);
    fclose(ostream);

    employee_free(e);

    return EXIT_SUCCESS;
}
```

重构后的代码与需求的描述更接近，虽然代码量膨胀了三倍，但能解决前文的问题：

1. 文件指针做空指针检查
2. 单链表容量能自动扩展
3. 字段类型或数目变化时仅修改输入和输出两处
4. 每项业务逻辑为独立的函数，易扩展且组合灵活

完整的代码请参见附录 I。

欢迎变化再次光临

经过重构的代码可维护性更好，因为每个函数的职责是单一的：

1. employee_read_node：应对输入源的变化，如列的顺序改变；
2. employee_read：应对集合结构的变化，如单向链表改成双向链表；

3. employee_print：应对输出格式的变化，如输出成 CSV 结构；
4. employee_adjust_salary：应对业务逻辑的变化，如调薪幅度增大。

不过，代码仍有不少重复之处，“重复”是维护性的大敌。想想你会如何应付下面这些问题？

数据源升级

上游系统在升级后，work.txt 的第一行提供了行数：

```
4
William 35 25000
Kishore 41 35000
Wallace 37 20000
Bruce 39 15000
```

而且，原系统频繁地申请空间也影响到性能。经过权衡，决定用数组取代单链表，这样只需一次性申请足够大的空间。

凭借面向对象风格的优势，对接口的实现的修改不会影响接口的使用，因此 main 函数无需任何修改。但对 Employee 对象而言却是灾难：每个接口的实现都与内部数据结构紧紧地绑在一起。几乎所有实现里都用 for 或 while 循环遍历整个链表，底层数据结构的变化意味着遍历方式的变化，即所有接口的实现全部需要重写！

优雅的访问文件

但凡涉及访问文件的代码，都需要 fopen、检查文件指针、存取数据、fclose，这几乎成了一种魔咒。比如 main 函数中，建立文件访问上下文的代码占去近一半的代码量。考虑规避这种魔咒，自动管理文件资源，在操作完成后自动关闭。

函数式风格

以上两个需求又足以让整个工程推倒重来。需求#1 要求抽象出遍历集合的方法，在迭代的过程中执行各自的循环体处理数据；需求#2 则要创建一种上下文，能自动打开文件，在执行访问操作完成后自动关闭。它们都涉及将代码块作为函数参数，在某个时刻调用，这正是函数式风格擅长的领域。

C 语言中，函数指针类型的变量可以指向参数类型与返回值类型都兼容的函数。虽然 C 不允许嵌套地定义函数或定义匿名函数，但确实允许将函数作为值传递，例如 qsort 的比较函数。

自定义遍历语句

先试着从 employee_print 和 employee_adjust_salary 中抽象出迭代过程：

```
typedef void (*Callback)(Employee);

void foreach(Employee e, Callback fn) {
   for (; e; e = e->next) {
      fn(e);
   }
}
```

其中 Callback 是自定义的函数指针类型，能接收一个 Employee 类型的参数，并且无返回值。

上述过程照搬了两个函数中相同的代码，但作为通用的迭代方法，这个实现有一个 bug：当 fn 的调用破坏了 e->next 的值时（例如调用 free），e = e->next 的值就变得未知。为了规避这个问题，需引入一个额外的变量：

```
void foreach(Employee e, Callback fn) {
   Employee p;
   while (p = e) {
      e = e->next;
      fn(p);
   }
}
```

在 fn 破坏节点内容前先获得 next 节点的引用，这样就能避免 free 这样具破坏性的过程影响遍历。使用这个自定义的控制语句（或高阶函数）重构 employee_free 函数，让它从遍历的细节中解放：

```
static void employee_free_node(Employee e) {
   if (e != NULL) {
      free(e);
   }
}

void employee_free(Employee e) {
   foreach(e, employee_free_node);
}
```

由于 C 不支持定义匿名函数，因此不得不定义一个释放单个节点的辅助函数。重构 employee_adjust_salary 与此类似：

```
static void employee_adjust_salary_node(Employee e) {
    if (e->salary < 30000) {
        e->salary += 3000;
    }
}

void employee_adjust_salary(Employee e) {
    foreach(e, employee_adjust_salary_node);
}
```

文件访问上下文

但是，重构 employee_print 的过程遇到了障碍：它需要一个额外的输出流，造成接口与 Callback 不兼容。似乎只能再为 IO 接口额外定义能接收两个参数的 IoCallback 接口，但如此一来又不得不实现一套专门处理它的 io_foreach，这是无法接受的！其实，利用偏函数技术能很优雅地解决这个问题，可惜 C 语言不允许定义匿名函数，也不支持闭包，只能感叹：臣妾做不到啊！由此可见匿名函数与闭包对函数式风格的友好性。

经过深思熟虑，我做出了一个很艰难地决定：使用 freopen 重定向标准输入输出流。这就能使用 printf 输出，无需提供额外的文件流。

```
void employee_print(Employee e) {
    for (; e; e = e->next) {
        printf("%s %d %d\n", e->name, e->age, e->salary);
    }
}
...
ostream = freopen("work.txt", "w", stdout);
...
employee_print(e);
```

如此，employee_print 的接口与 Callback 也兼容了，可用 foreach 来重构：

```
static void employee_print_node(Employee e) {
    printf("%s %d %d\n", e->name, e->age, e->salary);
}

void employee_print(Employee e) {
    foreach(e, employee_print_node);
}
```

有了以上基础，创建上下文的方法就呼之欲出了：

```
void with_open_file(String filename, String mode, Callback fn, Employee e) {
   File file = freopen(filename, mode, (mode[0] == 'r'? stdin: stdout));
   if (file == NULL) {
      fprintf(stderr, "Cannot open %s with %s mode.\n", filename, mode);
      exit(EXIT_FAILURE);
   }
   fn(e);
   fclose(file);
}
```

先把重定文件流到标准输入或输出流；执行回调函数；关闭文件流。如此，保存数据到文件的代码将简化成一句话：

```
with_open_file("work.txt", "w", employee_print, e);
```

对我而言，这样的代码很优雅，我迫不及待地希望 employee_read 也支持这种方式！但 employee_read 又是一块难啃的骨头：它不仅参数类型与 Callback 不兼容，连返回值类型也不同。为将返回值重构成 void，不得不提供一个额外参数保存返回值，并且类型是 Employee*：

```
static void employee_read_node(File istream, Employee* head) {
   Employee e = NULL;
   e = *head = (Employee)calloc(1, sizeof(struct _Employee));
   if (e != NULL && fscanf(istream, "%s%d%d", e->name, &e->age, &e->salary) != 3) {
      employee_free(e);
      *head = NULL;
   }
}

void employee_read(File istream, Employee* head) {
   Employee e = NULL, tail = NULL;

   *head = NULL;
   while (employee_read_node(istream, &e), e) {
      if (*head != NULL) {
         tail->next = e;
         tail = e;
      } else {
         *head = tail = e;
      }
   }
}
```

看起来这样就可以使用与 employee_print 相同的技巧去除 istream 这个参数了。其实不然，Callback 的参数是 Employee，不是 Employee*，接口依旧不兼容。这也是静态类型对函数式风格不友好的一个例子，静态类型在编译期就确定变量的类型，限制越多则灵活性越差，相应的受众面也越窄。

当然，C 语言也提供了一个替代方案，使用万能指针 void*作为 Callback 的参数（例如 qsort 就是这么做的）。但这样做，要么所有实现都要改成 void*，然后在函数里使用强制转换；要么得忍受编译器一堆类型不匹配的 warning。权衡再三，还是决定让 employee_read 牺牲小我，Callback 接口继续使用 Employee 做参数类型，在 employee_read 中将参数类型强制转换成 Employee*。

```
static void employee_read_node(Employee node) {
    Employee e = NULL, *head = (Employee*) node;
    e = *head = (Employee)calloc(1, sizeof(struct _Employee));
    ...
}

void employee_read(Employee list) {
    Employee e = NULL, *head = (Employee*) list, tail = NULL;

    *head = NULL;
    while (employee_read_node((Employee)&e), e) {
        ...
    }
}
```

解决问题

重构后的主函数变得愈加简洁，没有啰嗦的文件操作，甚至可以看成描述原始需求的伪代码。

```
int main(void) {
    Employee e = NULL;
    with_open_file("work.txt", "r", employee_read, (Employee)&e);
    employee_print(e);
    employee_adjust_salary(e);
    employee_print(e);
    with_open_file("work.txt", "w", employee_print, e);
    employee_free(e);
    return EXIT_SUCCESS;
}
```

重构后的完整代码请参见附录 II。回到需求#1，将单链表切换成数组，“struct _Employee *next”需替换成“int length”，employee_read 亦随之变化：

```
void employee_read(Employee list) {
   Employee e = NULL;
   int size;

   scanf("%d", &size);
   *((Employee*) list) = e = (Employee)calloc(size, sizeof(struct _Employee));
   e->length = size;
   foreach(e, employee_read_node);
}
```

与之前逐个为对象申请不同，现在一次性申请，即简化了代码又提高了性能。由于内存申请被转移出，employee_read_node 也得到极大的简化：

```
void employee_read_node(Employee e) {
   scanf("%s%d%d", e->name, &e->age, &e->salary);
}
```

因为空间已提前申请好，因此无需传入指针。伴随空间申请方式的改变，空间释放的方式也要相应调整：

```
void employee_free(Employee e) {
   free(e);
}
```

不再需要辅助函数 employee_free_node。接着是遍历：

```
void foreach(Employee e, Callback fn) {
   int i, length = e->length;
   for (i = 0; i < length; i++) {
      fn(e++);
   }
}
```

同样需要额外的变量 length 保存最初长度的信息。最后，还有一个可能意想不到的改动——employee_print。前文提过，“输出”是“输入”的逆操作，它的职责除了展示和保存数据，还要保持格式与输入兼容，即输出的数据还能再次被输入处理。因此需要在开头输出行数：

```
void employee_print(Employee e) {
   printf("%d\n", e->length);
   foreach(e, employee_print_node);
}
```

修改后的代码不仅实现了需求，而且变得愈加简洁！重点是无需修改业务处理的代码，因此业务逻辑也繁杂，函数式风格的优势越明显。完整代码请参见附录 III。

轮到你了！

客户对这次重构非常满意！这一回，他们希望 foreach 能改成并行，即每个循环体都在独立的线程中执行，那效率又会得到飞跃。

现在轮到你了，你会如何实现客户的需求？

附录 I：面向对象风格代码

```
#include <stdlib.h>
#include <stdio.h>

typedef char String[32];
typedef FILE* File;

typedef struct _Employee {
    String name;
    int age;
    int salary;
    struct _Employee *next;
} *Employee;

/* Destructor */
void employee_free(Employee e) {
    Employee p;
    while (p = e) {
        e = e->next;
        free(p);
    }
}

/* Input */
static Employee employee_read_node(File istream) {
    Employee e = (Employee)calloc(1, sizeof(struct _Employee));
    if (e != NULL && fscanf(istream, "%s%d%d", e->name, &e->age, &e->salary) != 3) {
        employee_free(e);
        e = NULL;
    }
    return e;
}

Employee employee_read(File istream) {
    Employee e = NULL, head = NULL, tail = NULL;

    while (e = employee_read_node(istream)) {
        if (head != NULL) {
            tail->next = e;
            tail = e;
        } else {
```

```
            head = tail = e;
        }
    }

    return head;
}

/* Output */
void employee_print(File ostream, Employee e) {
    for (; e; e = e->next) {
        fprintf(ostream, "%s %d %d\n", e->name, e->age, e->salary);
    }
}

/* Business Logic */
void employee_adjust_salary(Employee e) {
    for (; e; e = e->next) {
        if (e->salary < 30000) {
            e->salary += 3000;
        }
    }
}

int main(void) {
    File istream, ostream;
    Employee e = NULL;

    istream = fopen("work.txt", "r");
    if (istream == NULL) {
        fprintf(stderr, "Cannot open work.txt with r mode.\n");
        exit(EXIT_FAILURE);
    }
    e = employee_read(istream);
    fclose(istream);

    employee_print(stdout, e);

    employee_adjust_salary(e);

    employee_print(stdout, e);

    ostream = fopen("work.txt", "w");
```

```
    if (ostream == NULL) {
        fprintf(stderr, "Cannot open work.txt with w mode.\n");
        exit(EXIT_FAILURE);
    }
    employee_print(ostream, e);
    fclose(ostream);

    employee_free(e);

    return EXIT_SUCCESS;
}
```

附录 II：函数式风格代码

```
#include <stdlib.h>
#include <stdio.h>

typedef char String[32];
typedef FILE* File;

typedef struct _Employee {
    String name;
    int age;
    int salary;
    struct _Employee *next;
} *Employee;

typedef void (*Callback)(Employee);

/* High Order Functions    */
void foreach(Employee e, Callback fn) {
    Employee p;
    while (p = e) {
        e = e->next;                    /* Avoid *next be changed in fn */
        fn(p);
    }
}

void with_open_file(String filename, String mode, Callback fn, Employee e) {
    File file = freopen(filename, mode, (mode[0] == 'r'? stdin: stdout));
    if (file == NULL) {
        fprintf(stderr, "Cannot open %s with %s mode.\n", filename, mode);
        exit(EXIT_FAILURE);
    }
    fn(e);
    fclose(file);
}

/* Destructor */
static void employee_free_node(Employee e) {
    if (e != NULL) {
        free(e);
    }
}
```

```
void employee_free(Employee e) {
   foreach(e, employee_free_node);
}

/* Input */
static void employee_read_node(Employee node) {
   Employee e = NULL, *head = (Employee*) node;
   e = *head = (Employee)calloc(1, sizeof(struct _Employee));
   if (e != NULL && scanf("%s%d%d", e->name, &e->age, &e->salary) != 3) {
      employee_free(e);
      *head = NULL;
   }
}

void employee_read(Employee list) {
   Employee e = NULL, *head = (Employee*) list, tail = NULL;

   *head = NULL;
   while (employee_read_node((Employee)&e), e) {
      if (*head != NULL) {
         tail->next = e;
         tail = e;
      } else {
         *head = tail = e;
      }
   }
}

/* Output */
static void employee_print_node(Employee e) {
   printf("%s %d %d\n", e->name, e->age, e->salary);
}

void employee_print(Employee e) {
   foreach(e, employee_print_node);
}

/* Business Logic */
static void employee_adjust_salary_node(Employee e) {
   if (e->salary < 30000) {
      e->salary += 3000;
```

```
    }
}

void employee_adjust_salary(Employee e) {
    foreach(e, employee_adjust_salary_node);
}

int main(void) {
    Employee e = NULL;

    with_open_file("work.txt", "r", employee_read, (Employee)&e);
    employee_print(e);

    employee_adjust_salary(e);
    employee_print(e);
    with_open_file("work.txt", "w", employee_print, e);

    employee_free(e);

    return EXIT_SUCCESS;
}
```

附录 III：改造成用数组的代码

```
#include <stdlib.h>
#include <stdio.h>

typedef char String[32];
typedef FILE* File;

typedef struct _Employee {
    String name;
    int age;
    int salary;
    int length;
} *Employee;

typedef void (*Callback)(Employee);

/* High Order Functions    */
void foreach(Employee e, Callback fn) {
    int i, length = e->length;
    for (i = 0; i < length; i++) {
        fn(e++);
    }
}

void with_open_file(String filename, String mode, Callback fn, Employee e) {
    File file = freopen(filename, mode, (mode[0] == 'r'? stdin: stdout));
    if (file == NULL) {
        fprintf(stderr, "Cannot open %s with %s mode.\n", filename, mode);
        exit(EXIT_FAILURE);
    }
    fn(e);
    fclose(file);
}

/* Destructor */
void employee_free(Employee e) {
    free(e);
}

/* Input */
void employee_read_node(Employee e) {
```

```
    scanf("%s%d%d", e->name, &e->age, &e->salary);
}

void employee_read(Employee list) {
    Employee e = NULL;
    int size;

    scanf("%d", &size);
    *((Employee*) list) = e = (Employee)calloc(size, sizeof(struct _Employee));
    e->length = size;
    foreach(e, employee_read_node);
}

/* Output */
void employee_print_node(Employee e) {
    printf("%s %d %d\n", e->name, e->age, e->salary);
}

void employee_print(Employee e) {
    printf("%d\n", e->length);
    foreach(e, employee_print_node);
}

/* Business Logic */
void employee_adjust_salary_node(Employee e) {
    if (e->salary < 30000) {
        e->salary += 3000;
    }
}

void employee_adjust_salary(Employee e) {
    foreach(e, employee_adjust_salary_node);
}

int main(void) {
    Employee e = NULL;

    with_open_file("work.array", "r", employee_read, (Employee)&e);
    employee_print(e);

    employee_adjust_salary(e);
    employee_print(e);
```

```
    with_open_file("work.array", "w", employee_print, e);

    employee_free(e);

    return EXIT_SUCCESS;
}
```

附录 IV：Common Lisp 的解决方案

从函数式风格重构的过程能体会到，如果 C 语言能支持动态类型，那就不必在 employee_read 中做强制转换；如果 C 语言支持匿名函数，亦不用写这么多小函数；如果 C 语言除了能读入整型、字符串等基础类型，还能只能读入数组、结构体等复合类型，就无需 employee_read 和 employee_print 等输入输出函数……

许多对函数式风格支持更好的编程语言（如 Python、Ruby、Lisp 等）已经让这些“如果”变成现实！看看 Common Lisp 的解决方案：

```
(defparameter e (with-open-file (f #P"work.lisp") (read f)))

(print e)

(dolist (p e)
   (if (< (third p) 30000)
      (incf (third p) 3000)))

(print e)

(with-open-file (f #P"work.lisp" :direction :output) (print e f))
```

尝试用你自己熟悉的编程语言解决这个问题，并评估它的可维护性。