CS100 Recitation 4

目录

- Homework 1 讲解
- Quiz 1 讲解
- 重定向与文件 IO
- 读入任意长度的字符串

重定向与文件 IO

输入流、输出流

“流” (stream) ：这些字符像水流一样，流过便不再回头。

`stdin`, `stdout` ： 标准输入流、标准输出流

- 当我们在终端运行一个程序的时候，在默认情况下， `stdin` 和 `stdout` 都被绑定到这个终端。
- `./program < my_input_file.txt` ： 将 `stdin` **重定向**到 `my_input_file.txt`
- `./program > my_output_file.txt` ： 将 `stdout` **重定向**到 `my_output_file.txt`
  - 如果这个文件不存在，就创建出来。
  - 如果这个文件存在，它原有的内容会被覆盖掉。
  - `./program >> my_output_file.txt` ： 追加，而非覆盖。
- `./program < testcases/3.in > output.txt`

`freopen`: 在代码中实现“重定向” (redirect)

```
freopen("testcases/3.in", "r", stdin);
freopen("output.txt", "w", stdout);
```

File access mode string	meaning	explanation
`"r"`	read	open a file for reading
`"w"`	write	create a file for writing
`"a"`	append	append to a file
`"r+"`	read extended	open a file for read/write
`"w+"`	write extended	create a file for read/write
`"a+"`	append extended	open a file for read/write

`freopen`: 在代码中实现“重定向” (redirect)

```
freopen("testcases/3.in", "r", stdin);
freopen("output.txt", "w", stdout);
```

事实上 `freopen` 能做的不止是给 `stdin` / `stdout` 重定向！

永远不要仅凭课件上的只言片语和例子学习标准库函数。

标准输入、输出流

`scanf`, `printf`, `getchar`, `putchar`, `puts` 这些函数都使用 `stdin` 和 `stdout` 。

例如，

- 如果在使用 `scanf` 前将 `stdin` 重定向到某个文件， `scanf` 就会从那个文件读入。
- 如果在使用 `puts` 前将 `stdout` 重定向到某个文件， `puts` 就会向那个文件写入。

文件 IO

```
FILE *infile = fopen("relative/path/to/my/input/file", "r");
int a, b;
fscanf(infile, "%d%d", &a, &b);
FILE *outfile = fopen("relative/path/to/my/output/file", "w");
fprintf(outfile, "%d\n", a + b);
fclose(infile);
fclose(outfile);
```

- `fopen` ：打开一个文件，获得一个 `FILE *` （文件指针）。
- `fclose` ：关闭一个文件。参数是一个 `FILE *` 。
- `fscanf` , `fprintf` , `fputs` , `fgets` , `fputc` , `fgetc` ：在原来的函数的基础上，多传一个 `FILE *` 类型的参数。
  - 如果传入 `stdin` / `stdout` ，它就和普通的版本一样了。

读入任意长度的字符串

想法

逐字符读入，用 `malloc` 开一块 buffer 存储当前读入的内容。

如果 buffer 不够大了：

- 分配一块更大的内存，
- 把原来的东西拷贝过来，
- 把旧的内存释放掉（忘记这一步将导致内存泄漏）。

问题：重新分配内存时，应该开多大的？

简单的想法

假设原有的 buffer 长度为 n 。

当我们读入第 n 个字符（注意，始终要给空字符预留一个位置）时，重新分配一块长度为 $n+1$ 的，并把前面的 $n-1$ 个字符都拷贝过来。

用这种方式连续读入 N 个字符，一共会发生多少次拷贝？

简单的想法

假设原有的 buffer 长度为 n 。

当我们读入第 n 个字符（注意，始终要给空字符预留一个位置）时，重新分配一块长度为 $n+1$ 的，并把前面的 $n-1$ 个字符都拷贝过来。

用这种方式连续读入 N 个字符，一共会发生多少次拷贝？

$$\sum_{i=1}^{N}(i-1)=\frac{N(N-1)}{2}=\Theta\left(N^2\right).$$

输入 N 个字符，居然需要 $\Theta\left(N^2\right)$ 次操作？

更好的想法

假设原有的 buffer 长度为 n 。

当我们读入第 n 个字符（注意，始终要给空字符预留一个位置）时，重新分配一块长度为 $2n$ 的，并把前面的 $n-1$ 个字符都拷贝过来。

接下来读入第 $n+1, n+2, ..., 2n-1$ 个字符时，都不需要重新分配内存，也不需要拷贝任何元素。

用这种方式连续读入 N 个字符，一共会发生多少次拷贝？

更好的想法

假设原有的 buffer 长度为 n 。

当我们读入第 n 个字符（注意，始终要给空字符预留一个位置）时，重新分配一块长度为 $2n$ 的，并把前面的 $n-1$ 个字符都拷贝过来。

接下来读入第 $n+1, n+2, ..., 2n-1$ 个字符时，都不需要重新分配内存，也不需要拷贝任何元素。

用这种方式连续读入 N 个字符，一共会发生多少次拷贝？（假设 $N = 2^M$）

$$\sum_{i=1}^{M} \left(2^i - 1\right) = 2^{M+1} - M - 2 = 2N - \log_2 N - 2 = \Theta(N).$$

它相当于关于 N 的一次函数，这就可以接受了。

实现

`read_string.c`

迷思

你真的阻止内存泄漏了吗？

```
void work(void) {
  char *content = read_string();
  do_something(content);
  // ...
}
```

`read_string` 动态分配了内存，但这个内存需要用户来释放。

迷思

你真的阻止内存泄漏了吗？

```
void work(void) {
  char *content = read_string();
  do_something(content);
  // ...
  // 忘记 free(content) ，就是内存泄漏
}
```

不就是 `free` 吗！我不会忘的。

你真的不会忘记 `free` 吗？

```c
void work(void) {
  char *content = read_string();
  // ...
  if (condition1) {
    // ...
    return;
  }
  for (/* ... */) {
    if (condition2) {
      // ...
      return;
    }
    while (condition3) {
      // ...
    }
  }
  // ...
}
```

你真的不会忘记 `free` 吗？

```
/* read the file mode in octal */
param = getfield(tf);
mode = cvlong(param, strlen(param), 8);

/* read the user id */
uid = numuid(getfield(tf));

/* read the group id */
gid = numgid(getfield(tf));

/* read the file name (path) */
path = transname(getfield(tf));

/* insist on end of line */
geteol(tf);
```

如果程序中充斥着这些默默分配内存的函数，内存泄漏将取得彻头彻尾的胜利！

问题

根本的问题是：**内存的分配和释放不由同一个人完成**

有没有一种叫做“字符串”的东西来自己管理好这个内存？

- 上个世纪人们已经把这个问题讨论完了：

  《C Traps and Pitfalls》《Ruminations on C++》

C 标准库的解决方案：干脆让用户自己创建存放结果的内存，把这个地址传进来

- ```
  char *strcpy(char *restrict dest, const char *restrict source);
  ```

C++ 的最伟大的发明之一 RAII 可以很好地解决这个问题。
  ```