CS205 · C/C++ Program Design

Matrix

Monad

October 30, 2022

目录

1 功能展示

1. 基本功能

```c
matrix *A = createMatrixEx((Size){2, 2});  // assert A != NULL
float arr[4] = {.1f, .1f, .0f, .2f};
loadMatrixFromArray(A, arr);  // 新建矩阵 A = {{.1, .1}, {0, .2}}

matrix *B = createMatrixEx((Size){2, 2});  // assert B != NULL
fillMatrix(B, .5f);  // 用 0.5 填充整个矩阵

matrix *C = createUnallocatedMatrix();
mulMatrix(C, A, B);  // 矩阵乘法 C = A * B
printMatrix(C);

deleteMatrixEx(A);
deleteMatrixEx(B);
deleteMatrixEx(C);
```

2. 语义化参数

   传入两个非独立参数的时候（例如行列坐标），使用复合结构作为参数，而并非两个 `int`，这样函数的意义会更加直观，并且不容易出错。

```c
matrix A;

// 使用:
createMatrix(&A, (Size){2, 2});

// 而不是:
createMatrix(&A, 2, 2);
```

3. 文件读写

   对于有时候训练一些模型，需要将训练好的模型保存到文件中，或者从文件中读取模型，这时候就需要文件读写的功能。

```c
matrix *A = createMatrixEx((Size){2, 2});
matrix *B = createUnallocatedMatrix();

fillMatrix(A, .5f);
saveMatrix(&A, "matrix.bin");
loadMatrix(&B, "matrix.bin");

assert(matrixEqual(&A, &B, 1e-5f));
```

4. 文档

   为了方便用户使用，这里用了 Doxygen 格式写了大量注释，并且可以直接生成文档。这样用户无论在 IDE 中编写还是在了解源码的时候，都可以很方便地查阅文档。

   而且文件中代码与注释的行数比例，将近 1:1。

```
$ cloc matrix.*
       2 text files.
       2 unique files.
       0 files ignored.

github.com/AlDanial/cloc v 1.90  T=0.01 s (281.0 files/s, 143298.9 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                                1            100             13            350
C/C++ Header                     1             70            410             77
-------------------------------------------------------------------------------
SUM:                             2            170            423            427
-------------------------------------------------------------------------------
```

5. 单元测试

   作为一个框架，它的正确性是非常重要的。因此，为了保证代码的正确性，这里使用了 Google Test 框架进行单元测试。而且单元测试的覆盖率可以达到 90% 以上，足以证明单元测试的覆盖面之广。

2 分析

2.1 设计原则

2.1.1 接口风格

作为一个框架，非常重要的一点是，要保持一致的设计原则，即保持接口风格一致，特别是错误处理的风格一致。这样用户在使用的时候，就不需要去记忆太多的接口，方便用户使用。

本 Project 的设计原则为：

1. 对于大多数函数，将第一个参数作为它的输出参数，然后第二个及以后的参数作为输入参数，然后返回值作为错误码。

   例如

   ```
   matrix *A = ..., *B = ..., *C = ...;
   int err = mulMatrix(C, A, B);
   assert(err == MATRIX_SUCCESS);
   ```

   就是在计算矩阵乘法 $C = A \times B$：A 和 B 相乘，并将结果保存到 C 中。

   这样做的动机，是因为错误码总得有个方式传出来，如果定义全局变量的话，就会有线程安全的问题，而且也不太好用。如果用 `int*` 作为参数将错误码传出的话，用户就需要先定义一个 `int` 变量，然后将它的地址传入函数，然后再检查它的值，需要分三步，十分地麻烦。

   所以处于上述考虑，这里就将错误码作为返回值返回。

2. 然后对于一些特殊的函数，比如 `minInMatrix`，如果返回值是错误码的话，就会显得十分地反直觉。

   对于这部分函数，就将错误码作为输出参数，然后返回值就是“正常”的返回值（如果出错了，就返回一些特殊值，例如 `NAN`）。如果用户不关心具体错误，允许用户将错误码参数设置为 `NULL`。

2.1.2 错误处理

除了接口的风格之外，库还需要与用户“划清界限”，即我们需要清楚地规定，哪些错误是用户的责任，哪些错误是库的责任。换句话说，就是库需要检查哪些错误。

因为库不可能将所有错误都检查出来，首先一个是不可能，例如用户可能会传入一个非法的指针（非 `NULL`），这是用户的责任，库是无法检查出来的。当然，如果你真的想检测的话，你可以手动维护一个表，记录库分配过的 `matrix` 的地址，然后每次调用函数的时候，都检查一下是否属于该几何。但是这样做的话，不

仅会增加库代码的复杂性，也会大大降低效率，有点鸭子睁眼——大可不必(duck 不 闭)了。并且如果用户手动分配内存的话，或者用户真的想搞事的话，那无论多牛的错误检查也神仙难救了。~~而且如果啥都检查的话，跟 C 的“trust user”的惯例也有点不搭~~。

所以，在这个 Project 中，我们只检查一些比较明显的错误，例如用户传入的指针是 NULL，或者矩阵的大小不匹配等等。对于一些比较隐晦的错误，检查的成本非常高，甚至有点舍本逐末了，就不检查了。

2.2 内存分配

因为矩阵的大小是不固定的，所以需要动态分配内存。并且为了支持超大矩阵，这里数组的大小使用 size_t 类型存储，即

```
typedef struct matrix {
    size_t rows;
    size_t cols;
    float *p_data;
} matrix;
```

扯远了，刚刚说道动态分配内存。而动态分配内存，就意味着从 matrix* 到具体的内容，隔了两层指针，如何处理这两层指针的分配和释放，是一个十分重要的问题。

如果直接接管两层指针的分配和释放，那么用户就不需要关心这些细节。

但是这样就会有一个弊端，因为我们是把输出作为输出参数传入，所以用户肯定需要一个有效的 matrix 指针来调用函数。那这时候，我们需要它里面的 p_data 是有效的吗？不一定。因为大部分的使用场景，用户一般都是临时新建一个 matrix，然后再将其传入到输出参数，即

```
matrix *A = ...;     // (1) 如何创建 matrix？(下面讨论)

mulMatrix(C, A, B);  // 这里假设 C 的大小不是想要的，需要重新分配一下内存
```

这时候，如果我们在 (1) 处分配了 p_data，那么在 mulMatrix 中，还要把它释放掉，再分配一次。这样就会出现一次不必要的内存分配和释放，会降低效率。

所以为了区分这些场景，这里设计了一些不同的函数来满足用户需求，分别是

函数	负责 matrix 本身	负责 p_data	备注
int createMatrix(matrix *mat, Size size)	✗	✓	初始化“matrix A;”这种局部变量
matrix* createMatrixEx(Size size)	✓	✓	
matrix* createUnallocatedMatrix(void)	✓	✗	上文所述使用场景
void deleteMatrix(matrix *mat)	✗	✓	与 createMatrix 配合使用
void deleteMatrixEx(matrix *mat)	✓	✓	与 createMatrixEx 配合使用

对于 createMatrixEx 和 createUnallocatedMatrix 而言，都是返回 matrix*。因为只要出错了，那么一定是 malloc 的问题，就没必要弄一个错误码了。而且直接返回指针在这里也是十分方便的。

对于 createMatrix，因为它是用于初始化局部变量的，所以矩阵需要作为参数传入，返回值就可以是错误码。

对于两个 delete 函数，返回值都是 void。因为如果输入参数是 NULL，那与 free 一样，不做任何操作。如果是野指针，也检查不出来。

此外，这些创建 matrix 的函数，都会设置好 matrix 中的变量的值（即使是 createUnallocatedMatrix，也会设置为全 0），不会出现 UB。

2.3 语义化参数

假设在一个函数中，有两个 int 参数，如果你不看这两个参数的名字，它们有可能是指一个坐标，也有可能是指一个大小——你猜不出来它究竟是什么。

所以我们可以给一些有特定意义的参数（组）定义一些复合变量，例如

```
typedef struct Size {
    size_t rows;
    size_t cols;
} Size;
```

然后调用的时候，我们就会写

```
createMatrix(&A, (Size){2, 2});
```

这样一眼就可以看出来 (Size){2, 2} 是表示大小是 2×2，可以让我们 “Express ideas directly in code” [1]。

当然，上面这个小例子可能看不出来，但是如果是一个复杂的函数，有很多参数，那么这种方式就会显得非常有用。比如说一个取子矩阵的函数：

```
int submatrix(matrix *dst, const matrix *src, Position start, Size size);
submatrix(B, A, (Position){1, 1}, (Size){2, 2});
// vs
int submatrix(matrix *dst, const matrix *src, size_t start_row, size_t start_col, size_t size_row,
              size_t size_col);
submatrix(B, A, 1, 1, 2, 2);
```

这样一对比，结果不言而喻，第一个版本的代码完胜，基本上一眼看上去就能 get 到（~~如果把 Size 再具体成 Span，那将绝杀~~）。对于第二个版本的代码，我们需要花时间去看它的参数是什么意思，而且还可能会猜错，搞不好会把那 4 个参数理解成子矩阵的两个角的坐标，而不是位置 + 大小。

2.4 错误处理

由于本次 Project 需要处理的错误比较简单，大部分都是检查参数的合法性。而且检查参数的合法性都是在函数开头，即使真的出错了，也不需要做清理工作，直接返回即可。

然后写着写着发现，由于指针参数太多，每次都要写一大堆 `if` 语句，很容易出错，而且代码也很难看，所以就用了宏来简化代码。

例如

```
#define CHECK_ARGUMENT_NOT_NULL(argument)                                          \
    if ((argument) == NULL) {                                                      \
        PRINT_WARNING("%s: got unexpected NULL argument: " #argument, __func__);   \
        return MATRIX_ARGUMENT_INVALID;                                            \
    }
```

这样，我们可以调用 `CHECK_ARGUMENT_NOT_NULL` 来检查参数是否为 `NULL`，不需要写一大堆 `if` 语句。

值得注意的是，这里用了一个 `PRINT_WARNING` 宏，来打印警告消息。这个宏会在 Debug 模式下打印消息，帮助用户调试；在 Release 模式下为了减小体积，就不打印消息了。

然后除此之外，也有一些错误，它发生的时候不需要清理资源，我们也可以用一个宏来简化这种操作。这个宏受到 Rust 的 `?` 运算符启发，它可以在错误发生的时候直接返回，并且这个运算符在早期是一个叫做 `try!` 的宏。所以这里就弄了一个名为 `TRY` 的宏，用于在不需要清理资源的情况下，直接返回错误。

```
#define TRY(expression) {           \
        int code = (expression);    \
        if (code != MATRIX_SUCCESS) \
            return code;            \
    }
```

使用例子如下

[1] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#p1-express-ideas-directly-in-code

```c
int copyMatrix(matrix *dst, const matrix *src) {
    CHECK_ARGUMENT_NOT_NULL(dst);
    CHECK_MATRIX_NOT_NULL(src);

    TRY(rawCreateMatrix(dst, src->rows, src->cols));

    loadMatrixFromArray(dst, src->p_data);
    return MATRIX_SUCCESS;
}
```

当然可惜的是，C 没有 RAII，不能复刻 `try!` 的完整功能。对于一些复杂度情况，只能手动写 `if` 来处理了。

2.5 文件存储

因为需要储存矩阵的场景还蛮多的，所以有必要写一个矩阵的读写支持。

然后受到一些网络包处理方式的启发，这里就用文件的前 16 个字节储存矩阵的行数和列数，然后接下来的字节就是矩阵的数据。对于更具体的实现，因为目前绝大部分平台都是用小端序，并且浮点数存储都是 IEEE 754 标准，所以就直接用 `fread` 和 `fwrite` 来读写数据了。

```c
fwrite(&mat->rows, sizeof(mat->rows), 1, f);
fwrite(&mat->cols, sizeof(mat->cols), 1, f);
fwrite(mat->p_data, sizeof(mat->p_data[0]), mat->rows * mat->cols, f);
```

2.6 Doxygen

对于函数的注释，经过对比之后，选择了 Doxygen。因为它可以被大部分的 IDE 识别，可以让用户在编写的时候看到文档，并且 Doxygen 也可以生成 HTML 文档和 LaTeX 文档，非常好用。

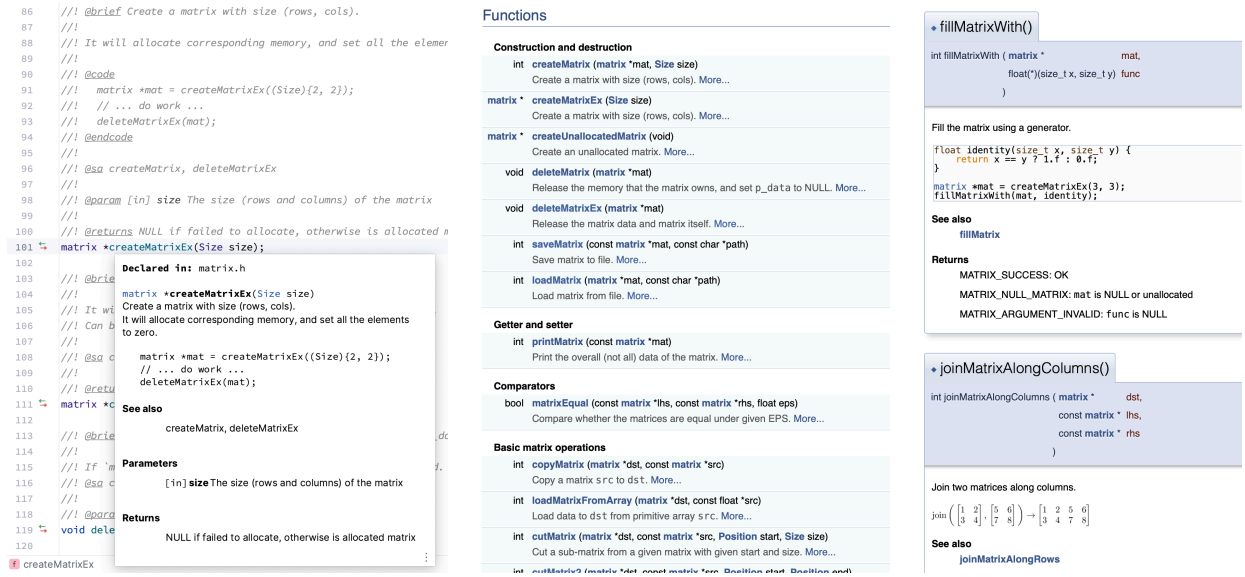
Doxygen 注释主要包括这几部分：

- `@brief`：函数的简要描述
- 函数的详细描述
- `@param`：参数描述
- `@returns`：返回值描述

写成注释，就是

```c
//! @brief (... brief description ...)
//!
//! (... detailed description ...)
//! @sa ( ... see also, if any ...)
//!
//! @param [out] param1 (... parameter description ...)
//! @param [in] param2
//! @returns (... return value description ...)
```

然后按照这样写注释之后，IDE 可以正确识别并弹出文档，而且用 `doxygen` 还能生成 HTML 文档：



(a) IDE popup (b) Doxygen 函数列表 (c) Doxygen 函数细节

Figure 1: Doxygen 效果

2.7 单元测试

前面也提到了，作为一个库，我们需要确保它的正确性。所以这里就用了 Google Test 来进行单元测试。然后为了衡量测试的覆盖面，这是用编译器的 `--coverage` 选项来编译，这样运行的时候就会生成覆盖率文件，然后再用 `lcov` 来生成覆盖率报告。

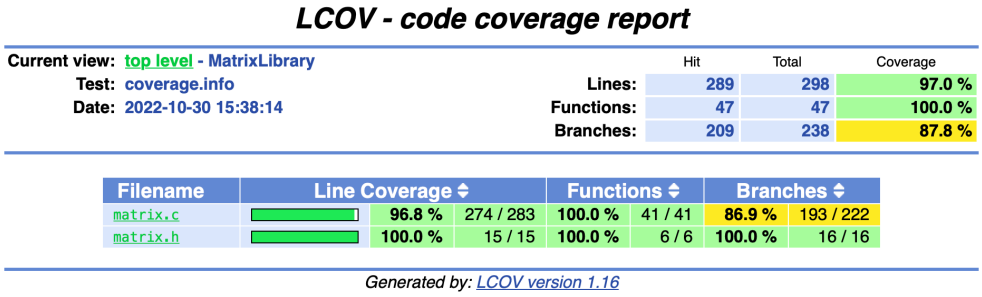


Figure 2: 覆盖率报告

可以看到，测试的代码覆盖率高达 96%，除去一些很难模拟的错误之外（比如 `malloc` 12 字节出错了），其它主要逻辑是几乎全覆盖的。

3 代码

代码已经与本报告一起提交在 Blackboard 上了，文件名是 `matrix.c` 和 `matrix.h`。然后毕竟具体代码有点长，也有点零碎，所以就不在报告中展示了。

然后按照传统，加上 `-Wall -Wextra` 编译（甚至还用了 `-Weverything`），本代码不会产生任何编译警告。

为了兼容旧编译器，本 Project 用的是 C99 标准，~~而且 C 的新标准也没有什么好用的东西~~。

然后这次 Project 如果把包括 `CMakeFiles.txt` 和测试代码在内的文件都提交了，那就有点太多了。所以这里就只提交了报告，还有 `matrix.c` 和 `matrix.h` 三个文件。对于其它构建文件和测试代码，可以在 GitHub `https://github.com/YanWQ-monad/SUSTech_CS205_Projects` 上找到，会在 ddl 截止之后放上去。

4 困难

4.1 inline 在 -O2 优化等级以下编译出错

问题描述

在 Clang 编译器下（毕竟 Apple 用的就是 Clang 编译器），如果用 gcc code.c 编译如下代码：

```c
inline int inline_func() {
    return 1;
}

int main() {
    return inline_func();
}
```

会发生编译错误：

```
$ gcc code.c
Undefined symbols for architecture arm64:
  "_inline_func", referenced from:
      _main in code-d6482f.o
ld: symbol(s) not found for architecture arm64
```

解决方法

这个问题是由于 Clang 在 C99 标准下对待 inline 的方式与 GNU C89 不同导致的。解决方法也很简单，在 inline 前面加上 static 就可以了。[2]

5 总结

因为这次的 project 只能用 C 来写，这样的话，像 reference、RAII、异常处理之类的好多好用的功能就用不了了。在这种情况下，就需要考验我们要在没有语法糖的情况下，如何写出一个好用的库。但是说实话，因为之前基本上没写过 C，所以也不知道 C 应该怎么写、有什么痛点，以及有什么好的解决方式。而且再加上这两周确实有点忙，没能好好打磨这次 project，所以这次 project 的效果可能不如前两次 project。

但是我也尽力在我的知识范围内，尽可能地在一些细节上处理好，包括内存管理不泄漏、覆盖率高的单元测试、较为完善的文档、语义化参数等等，不过对于如何用 C 语言写一个用户写得舒服的库，我确实没有什么能让人眼前一亮的点子。

最后按照惯例，这个项目也会放到 GitHub 开源，会放在 https://github.com/YanWQ-monad/SUSTech_CS205_Projects 仓库的 Project3 的子目录下。欢迎大家 star，谢谢！

[2] http://clang.llvm.org/compatibility.html#inline