

编写规范的 C 算法程序

文档类型	教学文档
编制	杜伟韬
版本	1.2
创建日期	05 年 12 月 25 日

引言

本文档来自于指导学生编写算法程序过程中总结出的经验，整理成文档便于他人参考，如有错误或不当之处，请发邮件至 duweitaotao@cuc.edu.cn

本文档及相关参考程序用于说明规范化的 C 算法程序如何编写，文中从以下几个方面探讨如何编写规范化的 C 算法程序

- 为什么要编写规范化的程序
- 文件层面的程序管理
- 代码层面的程序组成
- 如何验证和调试算法程序
- 文档的内容结构

本文档及相关的 demo 程序及算法说明书的编写目的是提供一个规模较小但尽量完整的示范性项目，让使用 C 编写算法程序的初学者能够快速入门，对于更多的编程格式规范，例如程序的版式、变量命名的规则等，文中没有一一论述，有兴趣的同学可以参考林锐博士编写的《高质量 C/C++ 编程》一文（目录 /refdoc 下）。

为什么要编写规范化程序

为什么对于一个算法程序而言，需要它具有规范化的特点呢？这主要是从程序的**可维护性**和**可重用性**方面所产生的需要，首先举一个系统维护的例子，当系统实现并调试完毕之后，系统工作正常，一年以后，用户发现在某些模式下系统不正常，于是研发人员重新调试代码，但由于时间较长现在已经看不懂当初的代码。

再举一个代码重用的例子，比如说，员工甲为当前的系统编写了一个算法程序 A，该程序在当前的系统中可以正常工作，但是过了一段时间（比如 4 个月之后）另外一个系统中也需要使用 A 算法，但是要对程序 A 进行调整，可能是接口上的调整（比如调整接口的参数），也可能是性能上的调整（比如允许该算法使用更多的内存从而提高运算速度），但是由于完成 A 算法时没有作文档记录，并且 A 程序中没有注释，另外员工甲已经记不得当时 A 程序中的算法细节，所以现在能作的只有根据算法的原理结合 A 程序的代码再次仔细的研究一遍代码中的机理，如果修改代码的任务仍旧由员工甲来完成，那么任务进展的还会顺利一些，但是更糟糕的情况可能是此时修改代码的任务由员工乙来完成，员工乙不了解 A 算法，于是他需要一方面研究 A 算法同时还需要研究没有任何注释和文档的 A 程序，在 A 算法规模较大的情况下，这会消耗非常多的时间。

从上文的例子可以看出，之所以需要编写程序说明文档和在代码中添加注释，其最主要的目的在于：

可维护性：便于程序的修改，包括接口的修改和性能上的修改

可重用性：可以使当前的算法程序在不同的系统中使用，比如一个在手机中使用的规范化的语音压缩程序，只要对其源代码进行较少的改动就可以在一个数字录音机中使用

那么我们应当如何编制这些规范化的算法程序呢？下面从几方面来讨论这个问题

文件层面

C 程序由头文件 *.h 和 *.c 文件构成（注：用于嵌入环境下的 C 程序所在的文件不要使用 cpp 作为扩展名，因为编译器可能会不支持 C++ 语言，另外使用 cpp 作为扩展名会使编译器对函数使用 C++ 的规则进行语法检查，比如可以在函数的任意处声明变量，这样会使得代码无法在一个仅支持 C 的编译器上编译）。以本文附带的 demo 程序为例，项目中包含 3 个文件，cfft16_TC.c，cfft16.c，cfft16.h 其中 cfft16_TC.c 是 fft 函数的调用者，cfft16.c 中包含一个 16 点的 fft 算法程序的实现代码，cfft16.h 中包含该算法程序的各种接口函数的声明，同时还包含有自定义的接口数据结构和数据类型的声明。

头文件 (*.h)

头文件用于代码模块之间的接口，头文件中保存接口函数的声明和接口数据类型的声明，头文件一般包括如下所示的 #ifndef 预编译宏，这样可以避免当头文件被 include 多次时所产生的函数和数据类型的重复定义

```
#ifndef A_H
#define A_H
    // function & data_type declaration
#endif
```

接口函数声明

接口函数在头文件中声明，其它的 *.C 文件如果要调用该接口函数只需要 include 声明该函数的头文件即可。

数据类型声明

接口函数需要使用的数据类型也要在头文件中定义，例如 demo 中的数据类型如下所示，这些数据类型的设计规则将在下文中解释。

```
typedef struct {
    cfft_t *inputBuf;
    cfft_t *turnBuf;
    cfft_t *twiddle;
    cfft4plan_t *cfft4planHandle;
} cfft16plan_t;
```

```
typedef struct {
    cfft_t *inputBuf;
} cfft4plan_t;
```

```
typedef struct {
    float r;
    float i;} cfft_t;
```

引用算法库函数

只需要 include 库函数的头文件，然后把算法库函数的 C 文件添加的项目中



思考一下，我们是如何使用 C 语言 stdio 和 math 中的库函数的？

C 源文件

C 源文件是以 .C 结尾的文件，其中主要包含以下几部分内容

函数的原型

指函数的实现代码

内部函数声明

通常一个 C 文件中会存在一些函数，这些函数仅被本文件中的其它函数调用，例如 demo 中 cfft.c 文件中的矩阵转置函数 turn()，这些函数的声明可以放在文件的开头，或是放在调用者函数原型的前面，例如 turn() 函数的声明位于 cfft16_proc() 函数原型之前。

常数表

某些情况下，需要在程序中保存一些常数表，这些数值可以用全局数组的形式存放在 C 文件中（此情况在 demo 中没有体现），如果在外部文件中引用这些全局变量的数值，可以使用 extern 关键字

上文描述了应当把程序的各种内容放在哪些文件里面，下面要描述的是程序本身应当如何设计才便于组织和维护。

代码层面

对于 C 代码而言，在设计和编写的过程中主要是在完成函数原型的设计和数据结构的设计。

函数原型

函数原型即函数的实现代码，算法程序主要有三种用途的函数它们是初始化函数、数据处理函数和析构函数。

初始化函数

初始化函数的作用是把需要使用的各种数据和存储空间准备好，例如 demo 中的初始化函数 cfft16_init()，其作用是申请需要用到的缓存，计算程序中需要使用的旋转因子系数。初始化函数通常不需要特别注重运行效率，例如 cfft16_init() 函数中计算旋转因子

的时候使用了 C 语言的库函数 `sin()` 和 `cos()`，因为初始化函数只在系统加载的时候运行一次，所以这种做法不会对性能造成影响。

数据处理函数

数据处理函数在系统运行中可能多次运行，所以编写此类函数的时候要注重效率，例如 demo 中的 `cfft4_proc()` 函数，它在 `cfft16_proc()` 被调用了 4 次，对于计算程序来说，衡量运算量的一个指标是乘法的次数，因为在各种处理器中乘法（相对于加减法和移位）是比较耗时的运算（除法和求模更加耗时），所以算法程序中尽量要少用乘法，并且尽量不要使用 `math.h` 中的数学库函数，虽然这些数学库函数在 PC 上的运行性能值得肯定，但是在其它的嵌入式平台上由于系统主频和系统资源以及库函数优化性能等原因它们表现的并不尽如人意。（注：由于涉及到众多的编译器、库函数和硬件细节，本文不过多讨论 C 程序的优化问题）

析构函数

析构函数主要用于释放申请的内存（例如 demo 程序中的 `cfft16_close()`，这些内存释放之后可以由其它的程序申请使用。

数据结构

数据结构是指程序的数据存储方案的设计。对于一个算法程序来说，它所需要数据可能包含以下几个部分

- 输入数据
- 输入的配置数据
- 输出的数据
- 计算的中间结果和各种常数表

所以，设计数据结构包括设计 函数的接口参数，以及 中间结果的封装形式，而函数的接口参数中又包括输入、输出数据和配置数据。

数据结构举例

例如，下面是一个商业化的音频编码器的初始化函数和编码函数的接口定义，在函数 `DrmencOpen()` 的变量列表中，参数 `hDrmEncoder *drmEncoder` 封装了编码器运行时的中间结果和常数表（因为要修改其值所以使用指针），其它的参数是编码器的配置参数。然后再看编码函数 `DrmencEncode()` 它的参数包括输入数据的起始地址 `*inbuffer`，输入数据的个数 `nSamples`，输出数据的起始地址 `*outbuffer`，输出数据的字节个数 `*nOutputBytes`（注：此参数值由编码函数来设定，所以要传指针），还有一个编码器的句柄 `hDrmEncoder drmEncoder`，这个句柄（handle）由初始化函数 `DrmencOpen()` 负责对其进行配置。

```
unsigned int DrmencOpen
(
    hDrmEncoder *drmEncoder,
    int audioCoding,
    int coderField,
    int coderSamplingRate,
    int audioMode,
    int stereoInputFlag,
    int SBROnOffFlag,
    int lengthOfAudioSuperFrame,
    int lengthHigherProtected
);
```

```
unsigned int DrmencEncode
(
    hDrmEncoder drmEncoderHandle,
    float *inbuffer,
    int nSamples,
    unsigned char *outbuffer,
    unsigned int *nOutputBytes
);
```

把上面音频编码器的例子中的设计思想应用到我们的 16 点 fft 函数设计中,把所有的中间结果和旋转因子表存放在结构体 `cfft16plan_t` 中,把输入的时域数据和输出的频域数据的地址指针添加到 fft 函数的入口参数中,于是得到了 16 点 fft 函数的原型声明,

```
int cfft16_proc(
    cfft16plan_t *cfft16planHandle,
    cfft_t *in,
    cfft_t *out);
```

至于结构体 `cfft16plan_t` 的设计,需要根据具体的 fft 实现算法来确定,在当前的算法中 16 点的 fft 是利用 4×4 行列分解的 4 点 fft 来实现的,根据这种算法的原理最后确定了 `cfft16plan_t` 中包括以下内容

```
typedef struct {
    cfft_t *inputBuf;    // 行列变换后输入数据的缓冲区
    cfft_t *turnBuf;     // 列 fft 后经过转置数据缓冲区
    cfft_t *twiddle;     // 旋转因子常数表
    cfft4plan_t *cfft4planHandle; // 4 点 fft 使用的结构体
} cfft16plan_t;
```

数据结构设计总结

这里的一个重要思想就是,把算法函数的中间结果缓冲区和常量数据表使用结构体封装起来,在初始化函数中对这些数据进行配置并申请内存。另外封装的时候尽量封装指针而不是对象,例如 `cfft16plan_t` 结构中封装了一个 `cfft4plan_t *` 类型的指针,这样作的好处是,即使被封装的子结构体拥有庞大的指针序列(例如,作为演示用途的 demo 中,结构体 `cfft4plan_t` 里面定义了许多 `noUsePointer` 指针),于是在封装该子结构体的父结构体中也只需要保存一份子结构体的指针而不是一个庞大的子结构体对象。



思考一下,父结构体中的这个指针应当指向哪里?如何初始化其目标实例?可以参考一下 demo 中 `cfft16_init()` 函数里的做法。

封装? or ! 封装?

为了计算 4 点 fft,在 demo 程序的数据结构中专门为 4 点 fft 定义了 `cfft4plan_t` 结构体用来封装其内部数据,通过这里我们可以看到,这种封装方法会让程序更加有层次化,从而便于维护,但是代码编写的工作量也会随之增大,如果不封装的话, demo 程序的数据结构可以是象下面一样,那么我们需要考虑的是在什么时候应当进行封装呢?

```
typedef struct {
    cfft_t *inputBuf16; // 16 点行列变换后输入数据的缓冲区
    cfft_t *turnBuf16;  // 16 点列 fft 后经过转置数据缓冲区
    cfft_t *twiddle16;  // 16 点旋转因子常数表
    cfft_t *inputBuf4;  // 4 点 fft 输入缓冲区
} cfft16plan_t;
```

一般来说如果是一个人的程序，在不影响程序的可读性的前提下，不必对程序进行过多的数据结构封装（2、3 层足矣），但是这种一个人编制的程序至少要对外作一层数据结构封装，从而便于用户的使用。

调试方法

对于算法程序而言，算法的设计可以使用 Matlab 软件进行，例如 demo 程序中 Msim 目录下的 fft16sim.m 文件，此文件中包含了使用旋转因子算法实现 16 点 FFT 程序的全部算法过程。使用 Matlab 软件作为算法设计工具的好处是：

- 1 可以利用 Matlab 丰富的功能函数，并且这些功能函数的正确性是可以保证的
- 2 可以把注意力集中在算法的设计上而不必考虑数据类型和存储格式的设计（这些都是 C 代码中要考虑的问题）
- 3 可以把 C 程序的运行结果和 Matlab 的运算结果逐步对比，从而快速的找出 C 程序中的错误。

调试用的预编译宏

可以使用宏开关来控制程序是否打印输出调试信息，利用这种宏可以实现程序的 Debug 版本和 Release 版本的控制，例如在 cfft16.c 文件中，程序中使用如下的代码段控制调试信息的输出。函数 ComplexPrint() 用于向标准终端 (stdout) 打印复数数组。

```
#ifdef DBG_CFFT16
    ComplexPrint("ColFFT", inputBuf, 16);
#endif
```

输出重定向

有的时候 C 程序中的算法函数的输出数据量较大，这是可以使用重定向的办法把输出到 stdout 的数据定向到一个文本文件中，例如在 VC 中设定：project—settings—Debug 标签—Program arguments 在其中填写 > result.m，其中 > 表示重定向，于是 C 程序中所有打印到 stdout 的字符都会被重定向到 result.m 文件中去。值得提一下的是在打印数据的时候可以使用 Matlab 的语法格式，例如 cfft16.c 文件中用于打印复数的函数 ComplexPrint()，该函数会使用 Matlab 的语法格式打印复数数据，这样做的好处是便于把 C 程序生成的数据直接拷贝到 Matlab 中进行验证。

Matlab 对比调试

当发现 C 算法程序中的结果不正确的时候，可以把 C 算法程序中的各个步骤的中间结果打印出来，然后和 Matlab 仿真程序的各个步骤的中间结果比对，这样可以快速的定位发生错误的位置。例如在 16 点的 FFT 程序中，算法分为输入数据转置、列 FFT、转置、旋转因子乘法、行 FFT，转置并输出数据几个步骤，这些步骤的结果均可以打印到文件中，然后和 Matlab 仿真程序中的各个步骤的结果进行比对。

如何编写程序文档

为程序编制文档是一个非常重要却又经常容易被忽视的过程，在程序设计的初始阶段常常由于进度的原因导致只有一些设计方案的草稿，而没有详细的文档。当程序设计及验证完毕后，应当为程序编写一个说明性的文档，用来描述程序的使用方法和设计原理，这样可以给别人方便的使用你编写的程序，并且该程序维护起来也会更容易，本教程的 demo/fft16/doc/目录下提供了一个示范性的文档，供大家参考。

一般来说程序文档包括如下几部分。

函数的使用说明，写给函数用户的文档，包括

- 函数功能的描述，对函数的功能进行一个总体的描述
- 函数的原型，列出函数的定义声明
- 函数的接口参数及说明，对函数接口的各个参数进行说明
- 接口参数的数据类型说明，如果使用了自定义的参数类型则需要对这些数据类型进行说明，例如 demo 中的复数类型

算法原理和数据结构的说明，用于程序维护的文档，包括

- 算法原理的描述，从数学上阐述清楚算法的原理
- 数据结构的描述，把 C 函数使用的内部数据结构描述清楚，因为所有的运算操作结果最终是保存到存储器中去的，所以描述清楚数据结构非常重要。数据结构的描述可以结合算法的原理进行。

额外说明:

对于文档设计可以使用这样的方法，并不是程序文档的所有内容均只能在程序设计完毕之后才能编写，程序设计和文档编写是一个交替进行的过程，例如 demo 所示的 FFT 程序及其文档，在 16 点的 FFT 算法（分解为 4×4 的旋转因子算法）确定下来之后就可以编写算法描述部分的文档草稿。在程序最终设计验证完毕之后，把所有过程中积累下来的文档草稿整理并排版便得到了最终的程序文档，这样编写文档的过程不但变得不枯燥，而且还对程序设计有良好的促进作用，并且可以帮助我们在程序设计时整理思路。一个好的办法是使用框图的形式，例如 demo 程序手册中描述数据结构的方式，其中的图片使用 SmartDraw 软件绘制。

总结

本文结合一个简单的 demo 程序，从几个层次说明了编写规范的 C 算法程序的各个方面需要注意的事项，并且对一个程序的整个设计流程进行了描述，希望本文对各位同学有所帮助，鉴于编写人员水平有限，如有不当之处请多提宝贵意见。



课后练习

- 1、 找一个你感兴趣的算法，仿照 demo 中的过程，使用 MATLAB 来设计算法的过程，然后使用 C 语言来实现该算法，最后使用 MATLAB 来验证 C 程序的正确性，不要忘记给你的程序编写说明书。当你做完这一切后，可以和同学或老师们交流一下（别忘了你有程序说明书的）。
- 2、 看一下目录\demo\example_lib\libfaad 中的代码，这是一个音频源解码器的算法库（AAC 压缩格式的解码器），其中有比较复杂的算法过程，同学们可以不了解具体的算法原理，只是学习它的文件组成结构和代码组成结构，从而对于大型算法程序的组织与结构产生一定的认识。例如研究一下：
 - 初始化函数是哪些，怎样初始化？
 - 析构函数是哪些，都做了哪些事情？
 - 数据处理函数是哪一个，它调用了哪些子函数？
 - 程序中定义了哪些结构体，函数之间是如何利用这些结构体交换数据的？

提示：该解码库的接口函数定义于 decoder.h 中，其各种结构体定义于 struct.h 中

修订历史

1.1	初始版本
1.2	根据教学情况，发现学生容易对数据进行过多层次的封装，添加“封装？or！封装？”一段