C 程 序 设 计

C Programming

2

代码阅读训练

附加课程

厦門大學

XIAMEN UNIVERSITY

信息学院 黄 炜

（国家示范性软件学院） 博士，副教授

School of Informatics Dr. Wei Huang

阅读代码
入门与提高的必由之路

阅读代码：入门与提高的必由之路

- 熟读唐诗三百首，不会吟诗也会吟
  - 计算机技术日新月异。程序员要不断地学习新的知识。
  - 计算机科学实践性强，许多内容在书本学不到。代码中凝聚着许多实践性的知识，通过阅读才能真正掌握真谛。
  - 工作的移交、新手的人门或是加人新的项目，都要阅读大量由他人编写的代码。
  - 不阅读代码，我们就会不断地重做别人已经完成的工作，重复过去已经发生过的成功和错误。

如何阅读代码

- 精读
  - 在每句阅读时，先理解每字的意思，然后通解一句之意，又 通解一章之意，相接连作去，明理演文，一举两得”这是传统的三步精读法。
- 速读
  - 对信息的筛选能力和筛选速度尤其重要。
- 读书笔记

怎么学习阅读大型项目的代码

- 你读的越多，你就越容易读懂
  - 因为高手写程序的思维都是趋同的，正所谓万剑归宗。
- 由上之下，逐步求精。
  - 不要一开始想把所有细节搞清，否则只见树木，不见森林。
- 调试。
  - 调试是程序员最重要的功底。
- 静下心来，耐心的调试，分析，总结，记得要记笔记。不断的假设、猜想，然后证实、证伪。

本节课目标

- 建立正确的代码阅读观念与态度
- 学习代码阅读的技巧
- 识别自己在代码阅读上的问题
- 学会代码阅读，提高工作效率
- 提升软件调试的效率

小窍门

- 养成习惯，经常花时间阅读别人编写的高品质代码。
- 有选择地阅读代码，同时要有自己的目标。
  - 学习新的模式、编码风格，还是满足某些需求的方法.
- 注意并重视代码中特殊的非功能性需求
  - 这些需求也许会导致特殊的实现风格
- 多数情况下，想要了解别人如何做，只有阅读代码
- 注意版权问题

小窍门

- 养成习惯，经常花时间阅读别人编写的高品质代码。
- 有选择地阅读代码，同时要有自己的目标。
  - 学习新的模式、编码风格，还是满足某些需求的方法.
- 注意并重视代码中特殊的非功能性需求
  - 这些需求也许会导致特殊的实现风格
- 多数情况下，想要了解别人如何做，只有阅读代码
- 注意版权问题

小窍门

- 在寻找bug时, 请从问题的表现形式到问题的根源来分析代码. 不要沿着不相关的路径(误入歧途).
- 要充分利用调试器和编译器给出的警告或输出
- 对于大型且组织良好的系统，只需最低限度了解全部功能，就能够对它做出修改
- 增加功能时，先找到实现类似特性的代码，作为模板
- 从特性的功能描述到代码的实现, 可以按照字符串消息, 或使用关 键词来搜索代码.

小窍门

- 在移植代码或修改接口时，您可以通过编译器直接定位出问题涉及的范围，从而减少代码阅读的工作量
- 重构时，从一个能够正常工作的系统开始做起，希望确保结束时系统能够正常工作。善用测试用例
- 代码重构时，先从系统的构架开始，然后逐步细化。
- 复查软件系统时，注意系统是由多部分组成的，不仅是执行语句。还要注意分析：文件和目录结构、生成和配置过程、用户界面和系统的文档。

代码阅读的工具

- 尽量借助于集成开发环境（IDE）
  - Visual Studio
  - Eclipse
- 版本控制工具
  - DIFF
- 不要用普通文本编辑器、代码跳转不健全的开发环境
  - 有现成工具不用，何必自找麻烦

代码阅读的方法与技巧

- 基本编程元素
- C数据结构
- 控制流程
- 文档
- 命名规范与约定

基本编程元素

- 全局变量和函数
- 了解函数的功能
  - 基于函数名猜测
  - 阅读位于函数开始部分的注释
  - 分析如何使用该函效
  - 阅读函数体的代码
  - 查阅外部的程序文档

基本编程元素

- 可选择的策略
  - 自底向上和自顶向下的分析
  - 应用试探法和检查注释和外部文档

C数据结构

- 程序的作用是将算法应用到数据之上。数据的内部组织对算法的执行至关重要。
- 许多不同的机制都可以将相同类型的元素组织成集合
  - 向量（数组）；矩阵和表；栈；队列；映射（map）
  - 集合（Set）；链表；树；图（graph）
- 根据底层的抽象数据类型，阅读直接的数据结构操作。

控制流程

- 顺序、分支和循环
- 递归、异常、异步、预处理

文档

- 意义重大的编码工作，或大型、有组织体制之下的项目，比如GNU和BSD，都会采纳一套编码规范、指导原则或约定。
- 计算机语言和编程系统为程序员如何表达一个给定的算法提供了大量的余地。代码规范提供风格上的指导，目标是增强代码的可靠性、易读性和可维护性。
- 在阅读代码时，规范和约定可以为您提供特定代码的额外指引，从而增加阅读的效率。

文档

- 需求规格说明
- 设计规格说明
- 测试规格说明
- 用户文档
- 文档的更新有可能是滞后的

生成文档的工具

- 文本文件
- **Markdown**
- 自动化生成工具
  - **doxygen**
  - **Javadoc**

命名规范与约定

- 文件的命名及组织
  - 遵守文件名和后缀的约定能够优化对源代码的搜索。
  - 许多文件名和扩展名跨不同项目和开发平台通用
- 缩进
  - 阅读代码时，先要确保编辑器和打印机的tab设置，与代码遵循的风格规范一致。

文档

- 文档对非功能性需求背后的理论基础—安全做了详尽的描述，对理解它在源代码中的实现很有帮助。
- 文档中，经常能够找到设计者当时的观点;系统需求的目标、目的和意图，构架，以及实现;还有当时否决的其他方案，以及否决它们的理由。
- 文档还会说明内部编程接口。大型系统会划分成多个子系统，子系统通过精确定义的接口进行互操作。
- 重要的数据集合经常组织为抽象数据类型或类。

命名规范与约定

- 编排
  - 空格和换行符(line break)的分布，花括号的放置。
  - 如果编写该段代码的程序员没有经过正确编排的训练，那么极有可能存在大量其他更严重的错误。
  - 不一致性也可能是多个程序员在同一段代码上工作的结果。
  - 编排上的冲突反映团队协调很差，程序员不尊重前辈工作。
  - 编码规范还规定了一些特殊的注释单词，使得所有的实现人员都能够容易地搜索和确认。

标识符的构造方式

- 首字母大写
  - 标识符中每个新单词的首字母都大写。
    - 例如：SetErrorMode，首字母是否大写要依附加规则而定。
- 下划线分隔单词
  - 标识符中每个附加单词都通过下划线与前面的单词分隔开
    - 例如：expanent_is_negatives。受Unix影响，GNU编码规范推荐。
- 只取首字母
  - 取每个单词的首字母合并组成新的标识符

如何命名常量

- 常量使用大写字母命名，单词用下划线分隔。
- 这种大写命名约定可能和宏互相冲突，因为宏也遵循相同的命令方案。

匈牙利命名记法

- 微软推广的一种关于变量、函数、对象、前缀、宏定义等各种类型的符号的命名规范。
- 主要思想是：在变量和函数名中加入前缀以增进人们对程序的理解。
- 有关项目的全局变量用g_开始，类成员变量用m_，局部变量若函数较大则可考虑用l_用以显示说明其是局部变量。
- 例如：lpsz表示long pointer string zero

编程实践

- 不同的编程规范对可移植构造的构成有不同的主张。
  - 例如，GNU和Microsoft编码指导原则都将可移植性的焦点放在处理器构架的不同上，并且认为特定的软件只会在GNU, Linux或Microsoft操作系统上运行。
- 审查代码的可移植性，或以某种给定的编码规范作为指南时，要了解规范对可移植性需求的界定与限制。
- 编码指导原则还经常规定应用程序应该如何编码以保证正 确或一致的用户界面。

过程规范

- 许多指导原则都为组织编译过程建立了精确的规则。
- 这些规则经常是基于特定的工具或标准化的宏
  - 熟悉了它们之后，您就能够很快地理解，遵循给定指导原则的任何项目的编译过程。
- 为了满足应用程序安装会对发布过程做出精确的规定。
  - Microsoft Windows installer使用的格式，或Red Hat Linux RPM格式，对组成发行包的文件类型、版本控制、安装目录和由用户控制的选项都有严格的规则。

大型项目代码阅读

- 了解文档结构，注意浏览与查阅方法使用
- 大型的项目远不只是编译后获得可执行程序
  - 一个项目的源码树一般还包括规格说明、最终用户和开发人员文档、测试脚本、多媒体资源、编译工具、例子、本地化文件、修订历史、安装过程和许可信息。
- 分析项目中的命名规范与约定
  - 粗略的浏览代码
  - 分析总结出代码的风格，命名规范等

应对大型项目

- 设计与实现技术
  - 大型的编码工作，由于其大小与范围，经常能够证明应用一些技术的必要性，而其他情况这些技术可能不值得使用。
- 项目的组织
  - 通过浏览项目的源代码树——包含项目源代码的层次目录结构，来分析项目的组织方式。
  - 源码树常常能够反映出项目在构架和软件过程上的结构。
- 编译过程和制作文件

应对大型项目

- **Make 和makefile**
  - 在使用诸如make之类的工具时，由文件的各个构成部分构造文件需要遵循的依赖关 系与规则在一个单独的文件中指定，一般命名为makefile或Makefile 。
  - make读取并处理该制作文件，之后发布适当的命令编译指定的目标。
  - 制作文件中绝大部分内容是一个个的项， 每个项描述目标、目标的依赖关系和从这 些依赖关系生成目标的规则。

配置

- 配置（configuration）
  - 允许开发者编译、维护和发展源代码的单一正式版本。
  - 只维护源代码的单一副本简单化更改和演化管理。
- 通过使用适当配置，相同源代码体可以:
  - 创建拥有不同特性的产品
  - 为不同的构架或操作条统构造产品
  - 在不同的并发环境下进行维护
  - 与不同的库链接

修订控制

- 修订控制系统可以跟踪代码的演化，标记 重大的事件，并记录更改背后的原原由， 允许我们查看和控制时间要素。

测试

- 设计良好的项目，都会预先为测试系统的全部或部分功能提供相应的措施。
- 这些措施可能隶属于一份经过深思熟虑，用来验证系统运作的计划，也可能是系统的开发者在实现系统的过程中实施的非正式测试活动的残余。
- 作为源代码阅读活动的一部分，我们首先应该能够识别并推理测试代码和测试用例，然后使用这些测试产物帮助理解其余的代码。

测试方法

- 断点调试和单步调试
- 日志(logging)调试
  - 开发人员用输出语句测试程序运作与他们的预期是否相同。
  - 程序的调试输出可以帮助我们理解程序控制流程和数据元素的关键部分。
  - 跟踪语句所在的地点一般是算法运行的重要部分。
  - 大多数程序中，调试信息一般输入到程序的标准输出流或文件中。

测试：宏

- 例： DEBUG宏
  - 用DEBUG宏来控制特定代码是否编译到最终可执行文件中。
  - 系统的产品版木一般在编译时不会定义DEBUG宏，从而略去了跟踪代码和它生成的输出

```
#ifdef DEBUG
  if (trace == NULL)
    trace = fopen("bgtrace", "w");
  fprintf(trace, "\nRoll: %d %d%s\n", D0, D1,
  race ? " (race)" : "");  fflush(trace);
#endif
```

测试：调试级别

- 调试级别(debug level)
  - 太多的跟踪信息可能会掩盖至关重要的信息，或者会将程序的运行速度降到根本没办法使用的程度。
  - 程序中常常定义一个称为调试级别(debug level)的整数，它用来过滤生成的消息。

```
if (debug > 4)
  printf("systime: offset %s\n", lfptoa(now, 6));
```

测试：断言

- 断言(assertion)
  - 在程序代码中测试各种在程序执行时应该为true的条件，称为断言(assertion)
  - 许多语言和库都能够在对程序结构影响最小情况下，测试断言，并在断言失败时，生成调试输出，常常终止程序。
  - 可以用断言来检验算法运作的步骤、函数接收的参数、程序的控制流程、底层硬件的属性和测试用例的结果。
  - C和C++库都使用assert进行断言。

回归测试

- 更有效率的测试形式是将程序的结果与已知正确的输出进行测试。
- 回归测试—通常基于早期经过人工检验的运行结果，经常在程序的实现被修复后执行，以确保程序中的各个部分没有被偶然地破坏。
- 回归测试一般由一个执行测试用例的测试工具、每个测试用例的输入、期望的结果和顺序执行所有用例的框架组成。

测试用例

- 在阅读源代码时，要记住：
  - 测试用例可以部分地代替函数规格说明。
  - 可以使用测试用例的输入数据对源代码序列进行预演。

总结

- 阅读features，以此来搞清楚该项目有哪些特性；
- 思考，想想如果自己来做有这些features的项目该如何构架；
- 下载并安装demo或sample。通过demo或sample直观地感受这个项目；
- 搜集能得到的doc，尽快地掌握如何使用这个项目；
- 如果有介绍项目架构的文档，通过它了解项目的总体架构，如果没有，通过api-doc了解源码包的结构；

总结

- 分两遍阅读源码。
  - 第一遍以应用为线索，以总体结构为基础，阅读在应用中使用到的类和方法，但不用过深挖掘细节，对于嵌套调用，只用通过函数名了解最上层函数的意义，这一遍的目的在于把大致结构了然于心。
  - 第二遍就是阅读类和方法的实现细节，以第一遍的阅读为基础， 带着疑问去阅读那些自己难以实现的模块。

总结

- 总结。回味这个项目设计上的精妙，用到了哪些设计模式，能在哪些领域可以借鉴等等。

如何写出整洁的代码

我们永远需要代码

- 不可能有机器将含糊的需求翻译为完美的程序
  - 编程人员要了解需求方
  - 机器要了解编程人员
- 混乱使得生产力接近于零
- 混乱的代码令人无处着手
- 编程中最多的动做不是打字而是来回阅读代码

有意义的命名

- 名副其实
  - 要做什么？常量变量有何意义？单位？

```
int d;  // eclapsed time, in day
if (x[0] == 4)
```

```
int ecalpsedTimeInDays, daysSinceCreation, fileAgeInDays;
```

- 避免误导
  - 避免易混淆的命名，避免单独的OIol和01

```
double nGroup;
double XYZControllerForEfficientHandlingOfThings;
double XYZControllerForEfficientStorageOfThings;
```

有意义的命名

- 做有意义的区分
  - 避免无意义冠词
  - 废话都是冗余

```
void copyString(const char src[], char dest[]);
struct Product;
int GetMoney();
```

```
void copyString(char a1[], char a2[]);
struct ProductData;
struct ProductInfo;
int dayValue, sizeVariable;
char * nameString;
int GetMoneyAmmount();
```

- 易于诵读的名称
  - 不要用自创的缩写

```
int GenDateYYMMDDHHIISS();
int DtaRcrd102();
```

```
int e = 4;
s += t[j] / 34 + 5;
```

```
int MAX_STRING_LENGTH = 100;
```

有意义的命名

- 使用可搜索的名称
  - 不要大量用数字常量，单字母名称
- 避免使用编码
  - 匈牙利命名法
  - 成员前缀
  - 接口和实现
- 避免思维映射
  - 避免自创规则，明确是王道

```
struct Point {
    double PointX;
    double PointY;
};
char * lpszName = "Hi";
struct Size {
    int m_nRows;
    int m_nCols;
};
```

```
int Add();
int GetAddress();
```

有意义的命名

- 类名

```
struct TaskManager;
```

  - 类名应该是名词或名词短语而不是动词

- 方法名

```
int IsNegative();
int GetSize();
void SetSize(int size);
```

  - 方法名应该是动词或动词短语

- 别扮可爱

```
void Kill();
```

```
void GoToHell();
```

- 每个概念对应一个词

```
const char * GetAddress();
const char * FetchName();
const char * RetrieveFullName();
struct DeviceManager;
struct DeviceController;
struct DeviceDriver;
```

  - 函数名应独一无二
  - 命名应一以贯之

有意义的命名

- 别用双关语

```
int Add(int a);
```

```
int Insert(int a);
int Append(int a);
```

- 尽量少用解决方案领域的名称
  - 你的读者是计算机专业

```
int JobQueue[JOB_QUEUE_SIZE];
int BalancingBooks[BOOK_SIZE];
```

  - 实在不行，应该使用所涉问题领域的名称
- 添加有语意的语境

```
const char * addrState;
const char * addrCity;
```

  - 但是不要添加无意义或者误导的语境

```
const char * state;
struct XMUAddress {
    const char * XMUAddrCity;
    ...
}
```

- 取名字最难的是良好的描述技巧和共同的文化背景

函数

- 函数务必短小
  - 函数不应该超过一屏
  - 超过了要考虑拆分：内聚出问题了
- 函数应内聚：只做一件事，做好这件事
  - 判断标准：是否还能再拆出一个函数
- 函数的语句应在同一个抽象层级上
  - 不同抽象层级：初始化棋盘函数调用、判断胜利的逻辑
- switch语句
  - 用多态代替switch语句

函数

- 使用描述性名称
  - 函数越短，功能越集中，越便于取名字
  - 长名称比短而令人费解的名称好
  - 好名字帮助理清思路，帮助改进
  - 追索好名称，导致代码改善重构
- 函数参数
  - 最理想的函数参数应该为0，参数越多用起来越麻烦
    - 有足够理由才能使用3个以上的参数
    - 内聚出问题了，考虑封装类

函数

- 标识参数
  - 说明你的函数不止做一件事

```
const char * getName(int isFullName);
```

```
const char * getFamilyName();
const char * getFullName();
```

- 动词与关键字
  - 函数参数的命名应该形成良好的动词名词对的形式。

```
void write(const char * Name);
void writeField(const char * Name);
int assertExpectedEqualsActual(int expected, int actual);
```

函数

- 无副作用
  - 副作用是一种谎言，违反了只做一件事的规则
  - 会对类中的变量做出未能预期的改动
  - 会导致时序性耦合和顺序依赖

```
void checkPassword(const char * account, const char * password) {
    const char * decPass = decrypt(password);
    if (strcmp(account, decPass)==0) {
        SessionInitialize();
        return true;
    }
    return false;
}
```

函数

- 无副作用
  - 输出参数：你可能需要一个类
    - 参数自然被视为输入参数，输出参数导致检查文档，中断思路
- 区分指令与询问
  - 将指令与询问区分开来，防止混淆的发生
- 区分异常和错误代码
  - 错误代码需要对应表，需要及时处理，不推荐
  - 错误处理就是独立的事

函数

- 别重复自己
  - 当你起心动念想要复制粘贴时，可能耦合出问题了
    - 例如：2048的上下左右走
    - 代码臃肿、修改困难
- 结构化编程
  - 函数每个模块只有一个出口一个入口
  - goto只有大函数才有道理，尽量避免使用
- 如何写出好的函数：先写代码、再打磨代码

注释

- 尽量不要对代码写注释
  - 不写注释的前提是让代码自己说话
  - 注释是弥补我们在用代码表达意图时遭遇的失败
  - 注释可能会撒谎（注释本身也是一种耦合）
- 注释不能美化糟糕的代码
- 用代码来阐述

```
// check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

```
if (IsEligibleForFullBenefits(employee))
```

注释

- 好注释
  - 法律信息

```
// Copyright (C) 2003-2012 ABCD Corporation
// Released under the terms of the GNU General Pubic License
```

  - 提供信息的注释

```
// return an instance of name being tested
const char * nameInstance();
```

    - 最好是代码提供

```
const char * nameInstanceBeingTested();
// format matched hh:mm:ss EEEE, MM dd, yyyy
Pattern timeMatcher = Pattern.compile(
    "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

  - 对意图的注释

```
// This is our best attempt to get a race condition
// by creating large number of threads
for ( i=0 ; i<2000 ; i++)
```

  - 阐释

```
if (strcmp(name[i], name[j])>0) // name[i]>name[j]
```

  - 警示

```
// NOTE: This function lasts one hour for huge array
void slowSort(int array[])
```

  - 待完成列表

```
// TODO: out of bound has not been checked
void bubbleSort(int array[])
```

注释

- 好注释
  - 放大
  - 为公共API写注释

```
// This repetition is really important. It removes
// starting space
strtrim(name);
```

- 坏注释
  - 越说越乱问题越多
  - 多余的注释

```
if (name==NULL)
    // null pointer means defaults are loaded
```

```
// check password given the password
int checkPassword(const char * password);
```

  - 误导性注释：不够精确的注释
  - 循规蹈矩的注释：为了自动生成帮助文档

注释

- 坏注释
  - 代码管理器

```
// Power by W Huang
// 2017.12.19 New function
void checkPointer(void * pointer)
```

    - 注释掉代码、归属与署名、日志注释
  - 废话注释

```
int dayOfMonth;  // the day of month
if (pointer) // normal
```

    - 能用函数或变量名时，勿用注释
  - 位置标记

```
//////// Check the pointer here ////////
if (pointer)
```

    - 代码会淹没在背景噪音中
  - 括号后的注释

```
if (pointer) {
    ...
} // if
```

注释

- 坏注释
  - HTML代码
  - 非本地信息：描述别人的函数
  - 信息过多
  - 不明显的联系
    - 和代码无关的注释
    - 注释本身还需要注释

格式

- 格式影响可维护性和可扩展性
- 垂直格式（参考报纸）
  - 文件行数：不要太长
  - 用空白行隔开：函数间、声明后
  - 概念相关的代码应当相近
    - 紧密联系的函数、依赖的函数、变量和调用
    - 调整顺序，删掉多余的注释
  - 变量应当靠近使用位置（即在最小作用域）内声明

格式

- 横向格式
  - 尽量不要超过80个字符
  - 空格字符将较弱的事物分开
    - 赋值操作符周围加上空白字符
    - 不要做没必要的对齐
  - 缩进：不要因为程序短就不回车违反缩进规则
  - 空范围
    - 除非空语句另起一行并处理缩进，否则很难引起察觉

对象和数据结构

- 数据尽量是私有的
- 以下不是C语言的内容

错误处理

- 使用异常而非返回代码
  - 返回代码需要及时处理
- 不要返回空值、不要传递空值
  - 返回空值给自己挖坑

```
if (name==NULL)
    return NULL;
```

```
if (name==NULL)
    return "";
```

边界

- 使用第三方代码
  - 程序包关注普适性，使用者注重特定需求的接口

单元测试

- TDD三定律
  - 在编写不能通过的单元测试前，不可编写生产代码。
  - 只可编码刚好无法通过的单元测试，不能编译也算不通过。
  - 只可编与刚好足以通过当前失败测试的生产代码。
- 测试也要整洁
  - 快速、独立、可重复、自足验证、及时

高级主题

- 代码简洁之道

C 程 序 设 计

C Programming

2

谢谢观看

附加课程

厦門大學

XIAMEN UNIVERSITY

信息学院 黄 煒

（国家示范性软件学院） 博士，副教授

School of Informatics Dr. Wei Huang