目录

1 编码必备 **2**
1.1 代码风格 . . . 2
1.1.1 中文排版 . . . 2
1.2 Google 风格 . . . 3
1.2.1 头文件 . . . 3
1.2.2 作用域 . . . 5
1.2.3 类 . . . 6
1.2.4 函数 . . . 7
1.2.5 Google 奇技 . . . 7
1.2.6 其他 C++ 特性 . . . 8
1.2.7 命名规则 . . . 10
1.2.8 注释 . . . 10
1.2.9 格式 . . . 11
1.2.10 特例 . . . 12
1.3 数论 . . . 12
1.3.1 欧几里得算法 . . . 12
1.3.2 扩展欧几里得算法 . . . 13
1.4 字符串 . . . 14
1.4.1 KMP 算法 . . . 14
1.5 树 . . . 15
1.5.1 二叉树先序遍历 . . . 15
1.5.2 二叉树中序遍历 . . . 16
1.5.3 二叉树后序遍历 . . . 17

2 c-mode **19**

3 c++-mode **19**

4 python-mode **19**

5 sh-mode **19**

6 go-mode **19**

7 rust-mode **19**

1 编码必备

1.1 代码风格

- 四个空格缩进，不要混用
- 一行中只放一条语句
  - if 一般需要换行，除非很简短的
  - if 有 else 分支，始终换行
  - while 一般换行，除非很简短
- 一行的长度不要超过 80 个字符
- 操作符前后一般都要添加一个空格
  - 单元运算符一般要紧跟操作对象
  - 标点符号后要一个
- 当语句为 if 或者 for、while 且只有一条执行语句时，可以省略大括号；当语句多于一条时或有 if、else 的嵌套时，则始终应该加上大括号。
- 关于大括号是否换行均可，但由于代码结构的原因，个人倾向于不换行

1.1.1 中文排版

- 我国国家标准要求弯引号，个人建议使用直角引号。
- 引号中再用引号使用双直角引号。
- 当引号表示讽刺、反语暗示时，使用弯引号（用法参考「西文排版」部分）。
- 省略号占两个汉字空间，包含六个点。
- 破折号占两个汉字空间。
- 点号（顿号、逗号、句号等）、结束引号、结束括号等，不能出现在一行的开头。
- 开始引号、开始括号、开始双书名号等，不能出现在一行的结尾。
- 句首字母大写。单词间留空格。
- 点号后加一个空格（如逗号、句号等）。
- 符号前不加空格的：度的标志、百分号等。

- 符号后不加空格的：货币标志、表正负数符号等。
- 符号后加空格：「at」标志（电子邮件除外）、版权标识、项目符号等。
- 括号、引号前后加空格，中间内容无空格。
- 连字符（-）将两个相关单词组合成一个单词
- 全角连接号（—）常表示文章中断、转折或说明
- 斜体的用法
  - 用来强调文中某个词或某句话。
  - 用来标记外来语以及读者不习惯的单词。
  - 文中出现的书名、剧名、美术作品的题目等等。
- 专有名词使用特定大小写。
- 标题可单用大写字母来排。
- 中英文之间需要加空格。
- 中文与数字之间需要加空格
- 中文与链接之间增加空格
- 专有名词使用特定大小写
- 使用正确的缩写
- 中英文混排使用全角标点
- 全角标点与英文或数字之间不加空格
- 遇到完整的英文句子使用半角标点

1.2 Google 风格

1.2.1 头文件

- 头文件应该能够自给自足 (self-contained, 也就是可以作为第一个头文件被引入), 以.h 结尾。至于用来插入文本的文件，说到底它们并不是头文件，所以应以.inc 结尾。不允许分离出 -inl.h 头文件的做法.
- 所有头文件都应该使用 #define 来防止头文件被多重包含, 命名格式当是: <PROJECT>_<PATH>_<FI
  .

- 尽可能地避免使用前置声明。使用 #include 包含需要的头文件即可。
- 只有当函数只有 10 行甚至更少时才将其定义为内联函数.
- 使用标准的头文件包含顺序可增强可读性, 避免隐藏依赖: 相关头文件, C 库, C++ 库, 其他库的.h, 本项目内的.h.

1. 译者 (YuleFox) 笔记

   (a) 避免多重包含是学编程时最基本的要求;

   (b) 前置声明是为了降低编译依赖，防止修改一个头文件引发多米诺效应;

   (c) 内联函数的合理使用可提高代码执行效率;

   (d) '-inl.h' 可提高代码可读性 (一般用不到吧:D);

   (e) 标准化函数参数顺序可以提高可读性和易维护性 (对函数参数的堆栈空间有轻微影响, 我以前大多是相同类型放在一起);

   (f) 包含文件的名称使用 ':' 和 '.:' 虽然方便却易混乱, 使用比较完整的项目路径看上去很清晰, 很条理, 包含文件的次序除了美观之外, 最重要的是可以减少隐藏依赖, 使每个头文件在 "最需要编译" (对应源文件处:D) 的地方编译, 有人提出库文件放在最后, 这样出错先是项目内的文件, 头文件都放在对应源文件的最前面, 这一点足以保证内部错误的及时发现了.

2. 译者（acgtyrant）笔记

   (a) 原来还真有项目用 '#includes' 来插入文本，且其文件扩展名 '.inc' 看上去也很科学。

   (b) Google 已经不再提倡 '-inl.h' 用法。

   (c) 注意，前置声明的类是不完全类型（incomplete type），我们只能定义指向该类型的指针或引用，或者声明（但不能定义）以不完全类型作为参数或者返回类型的函数。毕竟编译器不知道不完全类型的定义，我们不能创建其类的任何对象，也不能声明成类内部的数据成员。

   (d) 类内部的函数一般会自动内联。所以某函数一旦不需要内联，其定义就不要再放在头文件里，而是放到对应的 '.cc' 文件里。这样可以保持头文件的类相当精炼，也很好地贯彻了声明与定义分离的原则。

   (e) 在 '#include' 中插入空行以分割相关头文件, C 库, C++ 库, 其他库的 '.h' 和本项目内的 '.h' 是个好习惯。

1.2.2 作用域

- 鼓励在.cc 文件内使用匿名命名空间或 static 声明. 使用具名的命名空间时, 其名称可基于项目名或相对路径. 禁止使用 using 指示（using-directive）。禁止使用内联命名空间（inline namespace）。
- 在.cc 文件中定义一个不需要被外部引用的变量时，可以将它们放在匿名命名空间或声明为 static 。但是不要在.h 文件中这么做。
- 使用静态成员函数或命名空间内的非成员函数, 尽量不要用裸的全局函数. 将一系列函数直接置于命名空间中，不要用类的静态方法模拟出命名空间的效果，类的静态方法应当和类的实例或静态数据紧密相关.
- 将函数变量尽可能置于最小作用域内, 并在变量声明时进行初始化.
- 禁止定义静态储存周期非 POD 变量，禁止使用含有副作用的函数初始化 POD 全局变量，因为多编译单元中的静态变量执行时的构造和析构顺序是未明确的，这将导致代码的不可移植。

1. 译者 (YuleFox) 笔记
   (a) ‘cc‘ 中的匿名命名空间可避免命名冲突, 限定作用域, 避免直接使用 ‘using‘ 关键字污染命名空间;
   (b) 嵌套类符合局部使用原则, 只是不能在其他头文件中前置声明, 尽量不要 ‘public‘;
   (c) 尽量不用全局函数和全局变量, 考虑作用域和命名空间限制, 尽量单独形成编译单元;
   (d) 多线程中的全局变量 (含静态成员变量) 不要使用 ‘class‘ 类型 (含 STL 容器), 避免不明确行为导致的 bug.
   (e) 作用域的使用, 除了考虑名称污染, 可读性之外, 主要是为降低耦合, 提高编译/ 执行效率.
2. 译者（acgtyrant）笔记
   (a) 注意「using 指示（using-directive）」和「using 声明（using-declaration）」的区别。
   (b) 匿名命名空间说白了就是文件作用域，就像 C static 声明的作用域一样，后者已经被 C++ 标准提倡弃用。
   (c) 局部变量在声明的同时进行显式值初始化，比起隐式初始化再赋值的两步过程要高效，同时也贯彻了计算机体系结构重要的概念「局部性（locality）」。
   (d) 注意别在循环犯大量构造和析构的低级错误。

1.2.3 类

- 不要在构造函数中调用虚函数, 也不要在无法报出错误时进行可能失败的初始化.
- 不要定义隐式类型转换. 对于转换运算符和单参数构造函数, 请使用 explicit 关键字.
- 如果你的类型需要, 就让它们支持拷贝 / 移动. 否则, 就把隐式产生的拷贝和移动函数禁用.
- 仅当只有数据成员时使用 struct, 其它一概使用 class
- 使用组合常常比使用继承更合理. 如果使用继承的话, 定义为 public 继承.
- 真正需要用到多重实现继承的情况少之又少. 只在以下情况我们才允许多重继承: 最多只有一个基类是非抽象类; 其它基类都是以 Interface 为后缀的纯接口类.
- 接口是指满足特定条件的类, 这些类以 Interface 为后缀 (不强制).
- 除少数特定环境外, 不要重载运算符. 也不要创建用户定义字面量.
- 将所有数据成员声明为 private, 除非是 static const 类型成员 (遵循常量命名规则). 处于技术上的原因, 在使用 Google Test 时我们允许测试固件类中的数据成员为 protected.
- 将相似的声明放在一起, 将 public 部分放在最前.

1. 译者 (YuleFox) 笔记

   (a) 不在构造函数中做太多逻辑相关的初始化;

   (b) 编译器提供的默认构造函数不会对变量进行初始化, 如果定义了其他构造函数, 编译器不再提供, 需要编码者自行提供默认构造函数;

   (c) 为避免隐式转换, 需将单参数构造函数声明为 ‘explicit‘;

   (d) 为避免拷贝构造函数, 赋值操作的滥用和编译器自动生成, 可将其声明为 ‘private‘ 且无需实现;

   (e) 仅在作为数据集合时使用 ‘struct‘;

   (f) 组合 > 实现继承 > 接口继承 > 私有继承, 子类重载的虚函数也要声明 ‘virtual‘ 关键字, 虽然编译器允许不这样做;

   (g) 避免使用多重继承, 使用时, 除一个基类含有实现外, 其他基类均为纯接口;

   (h) 接口类类名以 ‘Interface‘ 为后缀, 除提供带实现的虚析构函数, 静态成员函数外, 其他均为纯虚函数, 不定义非静态数据成员, 不提供构造函数, 提供的话, 声明为 ‘protected‘;

(i) 为降低复杂性, 尽量不重载操作符, 模板, 标准类中使用时提供文档说明;

(j) 存取函数一般内联在头文件中;

(k) 声明次序: ‘public‘ -> ‘protected‘ -> ‘private‘;

(l) 函数体尽量短小, 紧凑, 功能单一;

1.2.4 函数

- 函数的参数顺序为: 输入参数在先, 后跟输出参数.
- 我们倾向于编写简短, 凝练的函数 (如果函数超过 40 行，可以简短).
- 所有按引用传递的参数必须加上 const.
- 若要使用函数重载, 则必须能让读者一看调用点就胸有成竹, 而不用花心思猜测调用的重载函数到底是哪一种. 这一规则也适用于构造函数.
- 只允许在非虚函数中使用缺省参数, 且必须保证缺省参数的值始终一致. 缺省参数与函数重载遵循同样的规则. 一般情况下建议使用函数重载, 尤其是在缺省函数带来的可读性提升不能弥补下文中所提到的缺点的情况下.
- 只有在常规写法 (返回类型前置) 不便于书写或不便于阅读时使用返回类型后置语法.

1.2.5 Google 奇技

- 动态分配出的对象最好有单一且固定的所有主, 并通过智能指针传递所有权.
- 使用 cpplint.py 检查风格错误.

1. 译者（acgtyrant）笔记

(a) 把智能指针当成对象来看待的话, 就很好领会它与所指对象之间的关系了.

(b) 原来 Rust 的 Ownership 思想是受到了 C++ 智能指针的很大启发啊.

(c) ‘scoped$_{\text{ptr}}$‘ 和 ‘auto$_{\text{ptr}}$‘ 已过时. 现在是 ‘shared$_{\text{ptr}}$‘ 和 ‘uniqued$_{\text{ptr}}$‘ 的天下了.

(d) 按本文来说, 似乎除了智能指针, 还有其它所有权机制, 值得留意.

(e) Arch Linux 用户注意了, AUR 有对 cpplint 打包.

1.2.6 其他 C++ 特性

- 所有按引用传递的参数必须加上 const.
- 只在定义移动构造函数与移动赋值操作时使用右值引用. 不要使用 std::forward.
- 若要用好函数重载，最好能让读者一看调用点（call site）就胸有成竹，不用花心思猜测调用的重载函数到底是哪一种。该规则适用于构造函数。
- 我们不允许使用缺省函数参数，少数极端情况除外。尽可能改用函数重载。
- 我们不允许使用变长数组和 alloca().
- 我们允许合理的使用友元类及友元函数.
- 我们不使用 C++ 异常.
- 我们禁止使用 RTTI.
- 使用 C++ 的类型转换, 如 $\text{static}_{\text{cast}}$<>(). 不要使用 int y = (int)x 或 int y = int(x) 等转换方式;
- 只在记录日志时使用流.
- 对于迭代器和其他模板对象使用前缀形式 (++i) 的自增, 自减运算符.
- 我们强烈建议你在任何可能的情况下都要使用 const. 此外有时改用 C++11 推出的 constexpr 更好。
- 在 C++11 里，用 constexpr 来定义真正的常量，或实现常量初始化。
- C++ 内建整型中, 仅使用 int. 如果程序中需要不同大小的变量, 可以使用 <stdint.h> 中长度精确的整型, 如 int16_{t}. 如果您的变量可能不小于 2^{31} (2GiB), 就用 64 位变量比如 int64_{t}. 此外要留意，哪怕您的值并不会超出 int 所能够表示的范围，在计算过程中也可能会溢出。所以拿不准时，干脆用更大的类型。
- 代码应该对 64 位和 32 位系统友好. 处理打印, 比较, 结构体对齐时应切记
- 使用宏时要非常谨慎, 尽量以内联函数, 枚举和常量代替之.
- 整数用 0, 实数用 0.0, 指针用 nullptr 或 NULL, 字符 (串) 用'\0'.

  整数用 0, 实数用 0.0, 这一点是毫无争议的.

  对于指针 (地址值), 到底是用 0, NULL 还是 nullptr. C++11 项目用 nullptr; C++03 项目则用 NULL, 毕竟它看起来像指针。实际上，一些 C++ 编译器对 NULL 的定义比较特殊，可以输出有用的警告，特别是 sizeof(NULL) 就和 sizeof(0) 不一样。

  字符 (串) 用'\0', 不仅类型正确而且可读性好.

- 尽可能用 sizeof(varname) 代替 sizeof(type).

  使用 sizeof(varname) 是因为当代码中变量类型改变时会自动更新. 您或许会用 sizeof(type) 处理不涉及任何变量的代码，比如处理来自外部或内部的数据格式，这时用变量就不合适了。

- 用 auto 绕过烦琐的类型名，只要可读性好就继续用，别用在局部变量之外的地方。
- C++11 , 任何对象类型都可以被列表初始化
- 适当使用 lambda 表达式。别用默认 lambda 捕获，所有捕获都要显式写出来。
- 不要使用复杂的模板编程
- 只使用 Boost 中被认可的库.

1. 译者（acgtyrant）笔记

   (a) 实际上，[缺省参数会改变函数签名的前提是改变了它接收的参数数量](http://www.zhihu.com/question/24439516/answer/27858964)，比如把 ‘void a()‘ 改成 ‘void a(int b = 0)‘, 开发者改变其代码的初衷也许是，在不改变「代码兼容性」的同时，又提供了可选 int 参数的余地，然而这终究会破坏函数指针上的兼容性，毕竟函数签名确实变了。

   (b) 此外把自带缺省参数的函数地址赋值给指针时，会丢失缺省参数信息。

   (c) 我还发现 [滥用缺省参数会害得读者光只看调用代码的话，会误以为其函数接受的参数数量比实际上还要少。](http://www.zhihu.com/question/24439516/answer/27896004)

   (d) ‘friend‘ 实际上只对函数 / 类赋予了对其所在类的访问权限，并不是有效的声明语句。所以除了在头文件类内部写 friend 函数 / 类，还要在类作用域之外正式地声明一遍，最后在对应的 ‘.cc‘ 文件加以定义。

   (e) 本风格指南都强调了「友元应该定义在同一文件内，避免代码读者跑到其它文件查找使用该私有成员的类」。那么可以把其声明放在类声明所在的头文件，定义也放在类定义所在的文件。

   (f) 由于友元函数 / 类并不是类的一部分，自然也不会是类可调用的公有接口，于是我主张全集中放在类的尾部，即的数据成员之后，参考 [声明顺序](https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/classes/#declaration-order) 。

   (g) [对使用 C++ 异常处理应具有怎样的态度?](http://www.zhihu.com/question/22889420) 非常值得一读。

(h) 注意初始化 const 对象时，必须在初始化的同时值初始化。

(i) 用断言代替无符号整型类型，深有启发。

(j) auto 在涉及迭代器的循环语句里挺常用。

(k) [Should the trailing return type syntax style become the default for new C++11 programs?](http://stackoverflow.com/questions/11215227/should-the-trailing-
讨论了 auto 与尾置返回类型一起用的全新编码风格，值得一看。

1.2.7 命名规则

- 函数命名, 变量命名, 文件命名要有描述性; 少用缩写.
- 文件名要全部小写, 可以包含下划线 (_) 或连字符 (-), 依照项目的约定. 如果没有约定, 那么 "_" 更好.
- 类型名称的每个单词首字母均大写, 不包含下划线: MyExcitingClass, MyExcitingEnum.
- 变量 (包括函数参数) 和数据成员名一律小写, 单词之间用下划线连接. 类的成员变量以下划线结尾, 但结构体的就不用, 如: $a_{localvariable}$, $a_{structdatamember}$, $a_{classdatamember\_}$.
- 声明为 constexpr 或 const 的变量, 或在程序运行期间其值始终保持不变的, 命名时以 "k" 开头, 大小写混合. 例如:
- 常规函数使用大小写混合, 取值和设值函数则要求与变量名匹配: MyExcitingFunction(), MyExcitingMethod(), $my_{excitingmembervariable}()$, $set_{myexcitingmembervariable}()$.
- 命名空间以小写字母命名. 最高级命名空间的名字取决于项目名称. 要注意避免嵌套命名空间的名字之间和常见的顶级命名空间的名字之间发生冲突.

  顶级命名空间的名称应当是项目名或者是该命名空间中的代码所属的团队的名字. 命名空间中的代码, 应当存放于和命名空间的名字匹配的文件夹或其子文件夹中
- 枚举的命名应当和常量或宏一致: kEnumName 或是 $ENUM_{NAME}$.
- 你并不打算使用宏, 对吧? 如果你一定要用, 像这样命名: $MY_{MACROTHATSCARESSMALLCHILDREN}$.
- 如果你命名的实体与已有 C/C++ 实体相似, 可参考现有命名策略.

1.2.8 注释

- 使用 // 或 /**, 统一就好.

- 在每一个文件开头加入版权公告.

  文件注释描述了该文件的内容. 如果一个文件只声明, 或实现, 或测试了一个对象, 并且这个对象已经在它的声明处进行了详细的注释, 那么就没必要再加上文件注释. 除此之外的其他文件都需要文件注释.

- 每个类的定义都要附带一份注释, 描述类的功能和用法, 除非它的功能相当明显.
- 函数声明处的注释描述函数功能; 定义处的注释描述函数实现.
- 通常变量名本身足以很好说明变量用途. 某些情况下, 也需要额外的注释说明.
- 对于代码中巧妙的, 晦涩的, 有趣的, 重要的地方加以注释.
- 注意标点, 拼写和语法; 写的好的注释比差的要易读的多.
- 对那些临时的, 短期的解决方案, 或已经够好但仍不完美的代码使用 TODO 注释.

1.2.9 格式

- 每一行代码字符数不超过 80.
- 尽量不使用非 ASCII 字符, 使用时必须使用 UTF-8 编码.
- 只使用空格, 每次缩进 2 个空格.
- 返回类型和函数名在同一行, 参数也尽量放在同一行, 如果放不下就对形参分行, 分行方式与函数调用一致.
- Lambda 表达式对形参和函数体的格式化和其他函数一致; 捕获列表同理, 表项用逗号隔开.
- 要么一行写完函数调用, 要么在圆括号里对参数分行, 要么参数另起一行且缩进四格. 如果没有其它顾虑的话, 尽可能精简行数, 比如把多个参数适当地放在同一行里.
- 您平时怎么格式化函数调用, 就怎么格式化列表初始化.
- 倾向于不在圆括号内使用空格. 关键字 if 和 else 另起一行.
- switch 语句可以使用大括号分段, 以表明 cases 之间不是连在一起的. 在单语句循环里, 括号可用可不用. 空循环体应使用 {} 或 continue.
- 句点或箭头前后不要有空格. 指针/地址操作符 (*, &) 之后不能有空格.
- 如果一个布尔表达式超过标准行宽, 断行方式要统一一下.
- 不要在 return 表达式里加上非必须的圆括号.

- 用 =, () 和 {} 均可.
- 预处理指令不要缩进, 从行首开始.
- 访问控制块的声明依次序是 public:, protected:, private:, 每个都缩进 1 个空格.
- 构造函数初始化列表放在同一行或按四格缩进并排多行.
- 命名空间内容不缩进.
- 水平留白的使用根据在代码中的位置决定. 永远不要在行尾添加没意义的留白.
- 垂直留白越少越好.

1.2.10 特例

- 对于现有不符合既定编程风格的代码可以网开一面.
- Windows 程序员有自己的编程习惯, 主要源于 Windows 头文件和其它 Microsoft 代码. 我们希望任何人都可以顺利读懂你的代码, 所以针对所有平台的 C++ 编程只给出一个单独的指南.

1.3 数论

1.3.1 欧几里得算法

1. 递归算法

```
int gcd(int a,int b){
    if(b==0) return a;
    else
        return gcd(b,a%b);
}
```

2. 非递归算法

```
int gcd(int a,int b){
    int x;
    while(b!=0){
        x=b;
        b=a%b;
        a=x;
    }
```

```
8       return a;
9   }
```

1.3.2 扩展欧几里得算法

在求最大公约数的同时，求出 $ax + by = gcd(a, b)$ 的一组解

1. 递归

```
1   int exgcd(int a,int b,int& x,int& y){
2       if(b==0){
3           x=1;
4           y=0;
5           return a;
6       }
7       int d=exgcd(b,a%b,x,y);
8       int z=x;
9       x=y;
10      y=z-y*(a/b);
11      return d;
12  }
```

2. 非递归

```
1   int exgcd(int a,int b,int& x,int& y){
2       int d;
3       x=1;
4       y=0;
5       int tmp;
6       while(b!=0){
7           tmp=b;
8           b=a%b;
9           a=tmp;
10          tmp=x;
11          x=y;
12          y=tmp-y*(a/b);
13      }
14      d=a;
```

```
15      return d;
16  }
```

1.4 字符串

1.4.1 KMP 算法

1. next 数组 表示第 i 位不匹配时，应该将模式字符串指针设为多少，KMP 算法充分的利用了已经匹配所用到的信息; 为 -1 表示需要改变字符串指针;

2. 算法代码

```
1  #include<iostream>
2  #include<vector>
3  #include<string>
4  using namespace std;
5  vector<int> getNext(string s) {
6      vector<int> ret(s.length(), -1);
7      int m=ret.size();
8      if (ret.empty()) return ret;
9      for (int i = 1, j=-1; i < m; ++i) {
10         while (j > -1 && s[j+1] != s[i]) j = ret[j];
11         if (s[j+1] == s[i]) ++j;
12         ret[i] = j;
13     }
14     return ret;
15 }
16
17 vector<int> kmp(string s1, string s2) {
18     vector<int> ret;
19     int sl1=s1.length(), sl2=s2.length();
20     if (!(sl1 && sl2)) return ret;
21     auto nv(getNext(s2));
22     for (int i=0, j = 0; i < sl1;) {
23         if (-1 < j && s1[i] == s2[j]) ++i, ++j;
24         else if (j < 0) j = 0, ++i;
25         else j = nv[j];
```

```
26          if (j == sl2) ret.push_back(i - j), j = nv[j - 1];
27      }
28      return ret;
29  }
```

1.5 树

1.5.1 二叉树先序遍历

1. 递归

```
1
2   //前序遍历
3   void preorder(TreeNode *root, vector<int> &path)
4   {
5       if(root != NULL)
6       {
7           path.push_back(root->val);
8           preorder(root->left , path);
9           preorder(root->right, path);
10      }
11  }
```

2. 非递归

```
1   // 非递归前序遍历
2   void preorderTraversal(TreeNode *root, vector<int> &path)
3   {
4       stack<TreeNode *> s;
5       TreeNode *p = root;
6       while(p != NULL || !s.empty())
7       {
8           while(p != NULL)
9           {
10              path.push_back(p->val);
11              s.push(p);
12              p = p->left;
13          }
```

```
        if(!s.empty())
        {
            p = s.top();
            s.pop();
            p = p->right;
        }
    }
}
```

1.5.2 二叉树中序遍历

1. 递归

```
//中序遍历
void inorder(TreeNode *root, vector<int> &path)
{
    if(root != NULL)
    {
        inorder(root->left, path);
        path.push_back(root->val);
        inorder(root->right, path);
    }
}
```

2. 非递归

```
// 非递归中序遍历
void inorderTraversal(TreeNode *root, vector<int> &path)
{
    stack<TreeNode *> s;
    TreeNode *p = root;
    while(p != NULL || !s.empty())
    {
        while(p != NULL)
        {
            s.push(p);
            p = p->left;
```

```
12          }
13          if(!s.empty())
14          {
15              p = s.top();
16              path.push_back(p->val);
17              s.pop();
18              p = p->right;
19          }
20      }
21  }
```

1.5.3 二叉树后序遍历

1. 递归

```

void postorder(TreeNode *root, vector<int> &path)
{
    if(root != NULL)
    {
        path.push_back(root->val);
        postorder(root->left, path);
        postorder(root->right, path);
    }
}
```

2. 非递归

```
//非递归后序遍历-迭代
void postorderTraversal(TreeNode *root, vector<int> &path)
{
    stack<TempNode *> s;
    TreeNode *p = root;
    TreeNode *q=nullptr;//刚刚访问过的结点
    while(p != NULL || !s.empty())
    {
```

```
9           while(p != NULL) //沿左子树一直往下搜索，直至出现没有左子树的结点
10          {
11              s.push(p);
12              p = p->left;
13          }
14          q=nullptr;
15          while(!s.empty())
16          {
17              p = s.top();
18              s.pop();
19              if(p->right == q)   // 右孩子不存在或者刚访问
20              {
21                  path.push_back(p->val);
22                  q=p;
23              }
24              else  //第二次出现在栈顶
25              {
26                  s.push(p)
27                      p = p->right;
28                  break;
29              }
30          }
31      }
32  }
```

2 c-mode

3 c++-mode

4 python-mode

5 sh-mode

6 go-mode

7 rust-mode