C++ Note

collated by decem17

/* */不能嵌套

文件结束符 Windows: Ctrl + Z　UNIX: Ctrl + D

Visual Studio 注释: Ctrl + K + C　　解注释: Ctrl + K + U

内置类型: int, char, double, bool…

字符串字面值 literal 的类型实际上是由常量字符构成的数组, 结尾处有一个空字符'\0'

变量 variable 和对象 object 一般可以互换使用

初始化 initialized 不是赋值 assign. 初始化是创建变量是赋予其一个初始值, 而赋值是把对象的当前值擦除, 以一个新值替代.

列表初始化 int a = 0; int a = {0}; int a{0}; int a(0);

函数体内部的内置类型变量不被初始化

若想声明一个变量而非定义它, 则在变量名前加 extern, 如 extern int a;

标识符 identifier 由字母, 数字和下划线组成, 其中必须以字母或下划线开头, 对大小写敏感

大多数作用域 scope 用{}分隔. 全局作用域 global scope; 块作用域 block scope, 可嵌套

复合类型 compound type: 引用 reference, 指针 pointer

引用本身不是一个对象

引用必须初始化, 无法将引用绑定到另一个对象上

指针是一个对象, 无须在定义时赋初值, 但建议初始化所有指针

空指针不指向任何对象. int* p = nullptr; int* p = 0; int* p = NULL; //需要#include cstdlib

const 对象必须初始化, 只能在 const 类型的对象上执行不改变其内容的操作

const 对象仅在文件内有效, 若想在多个文件中共享该对象, 必须在变量的定义前加 extern, 如: extern const int a = 0; //源文件 extern const int a; //头文件

对常量的引用 reference to const: 把引用绑定到 const 对象上, 如:
const int a = 0; const int& r1 = a; //正确, r1 和 a 都是常量 r1 = 1; //错误 int& r2 = a; //错误, 非常量引用不能指向一个常量对象

允许为一个常量引用绑定非常量的对象, 字面值或表达式, 如:
int a = 0; const int& r1 = a; //正确(但 r1 = 0; //错误, 不允许通过 r1 改变 a 的值) const int& r2 = 0; //正确 const int& r3 = r1 * 2; //正确 int& r4 = r1 * 2; //错误, 非常量引用不能指向一个常量对象

指向常量的指针 pointer to const: const int* p = &a; //*p 不可变, 但 p 可改变指向

常量指针 const pointer: int* const p = &a; //*p 可变, 但 p 永远指向 a. 常量指针必须初始化.

顶层 const(top-level const)表示指针本身是个常量, 更一般地, 表示任意的对象是常量, 如算数类型, 类, 指针等.

底层 const(low-level const)表示指针所指的对象是个常量, 更一般地, 与指针和引用等复合类型的基本类型部分有关, 如:
int a = 0; int* const p1 = &a; //顶层 const, p1 值不可变 const int b = 1; //顶层 const, b 值不可变
const int* p2 = &b; //底层 const, p2 值可变 const int* const p3 = p2; //右为顶层 const, 左为底层 const.
const int& r = b; //用于声明引用的 const 都是底层 const

当执行拷贝操作时, 顶层 const 不受影响: a = b; p2 = p3; //p2, p3 指向类型相同, 忽视 p3 顶层 const 底层 const 则有所限制, 拷入和拷出的对象必须具有相同的底层 const 资格, 或者两个对象的数据类型可转换. 一般来说, 非常量可转换成常量, 反之则不可: int* p = p3; //错误, p3 有底层 const 定义而 p 没有 p2 = p3; //正确, p2 和 p3 都是底层 const. int& r = b; //错误, 普通的 int&不可绑定到常量上
const int& r2 = a; //正确, const int&可绑定到非常量上

常量表达式 constexpr 是指值不会改变并且在编译过程中就能得到计算结果的表达式, 如:
constexpr int a = 0; constexpr int b = a + 1; constexpr int c = size(); //只有当 size 是一个 constexpr 函数时才是一条正确的语句

constexpr 函数中: 函数的返回类型及所有形参的类型都是字面值类型, 而且函数体中必须有且只有一条 return 语句, 如: constexpr int size() {return 10;} 允许 constexpr 函数的返回值不是一个常量.

字面值类型 literal type 是声明 constexpr 时用到的类型. 包括算术类型, 引用和指针. 不包括自定义类, IO 库, string 类型, 因此不能被定义成 constexpr.

一个 constexpr 指针的初始值必须是 nullptr, 0, 或是存储于某个固定地址中的对象.

类型别名: 1. typedef double db; //db 是 double 的别名
2. using SI = Sales_item; //Sales_items 是一个自定义类

注意: 如果某个类型别名指代的是复合类型或常量, 会产生意想不到的后果, 如:
typedef char* pstring; const pstring cstr = 0; //cstr 是指向 char 的常量指针 const pstring* ps; //ps 的对象是指向 char 的常量指针

若 decltype 的表达式不是一个变量, 则返回表达式结果对应的类型, 如:
int a = 0, &r = a; decltype(r)返回引用类型, 但 decltype(r + 0)返回 int 类型.

若 decltype 的表达式内容是解引用, 则返回引用类型, 如:
int* p = &a; decltype(*p)返回 int&类型而不是 int.

若 decltype 使用的是一个不加括号的变量, 则得到该变量的类型. 若给变量加上括号, 编译器就会把它当成表达式, 所以会返回引用类型, 如: decltype(a)返回 int, decltype((a))返回 int&.
切记: decltype((variable))的结果永远是引用, 而 decltype(variable)的结果只有当 variable 本身是引用时才是引用.

头文件通常包含只能被定义一次的实体, 如类, const 和 constexpr 变量.

string 的初始化: string s1; //s1 是一个空字符串 string s2 = "hello"; sting s3("hello"); //s2, s3 是该字面值的副本 string s4(3, 'c'); //s4 的内容是"ccc" string s5(s4) //s5 是 s4 的副本

string 对象会自动忽略开头的空白(空格, 换行, 制表等)并从第一个真正的字符开始读起, 直到遇见下一处空白为止. 若 cin 的是" hello ", 则 string 保存的是"hello".

读取未知数量的 string 对象: string word; while (cin >> word) {cout << word << endl;} //每个单词后跟一个换行

getline 函数: 保留输入时的空白符, 遇到换行符就返回结果, 但不包括换行符. 如:
string line; while (getline(cin, line)) {cout << line << endl;}

字面值和 string 对象相加, 必须确保每个+两侧至少有一个是 string, 如:
string s1 = "hello"; string s2 = s1 + "world"; //正确 string s3 = "hello" + "world"; //错误
string s4 = s1 + "hello" + "world"; //正确
字符串字面值并不是 string 的对象, 它们是不同的类型.

一般来说, C++程序应使用名为 cname 的头文件而不是 name.h 的形式

基于范围的 for 语句 for (declaration : expression) {statement} 如: string str("abcde"); for (auto c : str) {cout << c << endl;} //每行一个字符 for (auto& c : str) {c = toupper(c);} //转换成大写

vector 是一个类模板 class template

不存在包含引用的 vector, 因为引用不是对象

初始化 vector 对象: vector<T> v1; vector<T> v2 (v1); vector<T> v3 (n, val); vector<T> v4 (n); vector<T> v5 {a, b, c …}

push_back 成员函数: 向 vector 对象中添加元素, 如: string word; vector<string> text;
while (cin >> word) {text.push_back(word);}

vector 对象和 string 对象的下标可用于访问已存在的元素, 而不能用于添加元素. 否则会产生缓冲区溢出 buffer overflow.

迭代器: begin / end 成员. begin 返回指向第一个元素的迭代器, end 返回指向尾元素的下一位置的迭代器(off the end). 如: string s("abcde"); for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it) {*it = toupper(*it);} //改写成大写形式

养成使用!=和迭代器的习惯, 就不用太在意用的是哪种容器类型.

cbegin(), cend()函数返回 const_iterator 类型.

it -> mem 等效于 (*it).mem

向量中间元素的迭代器: auto mid = v.begin() + (v.end() – v.begin()) / 2

字符数组: char a1[] = {'C', '+', '+'}; //维度为 3 char a2[] = {'C', '+', '+', '\0'}; //维度为 4 char a3[] = "C++"; //维度为 4 char a4[3] = "C++"; //错误

数组不能拷贝和赋值给其他的数组. 但允许使用数组初始化 vector 对象, 如:
int arr[] = {0, 1, 2}; vector<int> ivec(begin(arr), end(arr));

使用数组时编译器一般会把它转换成一个指向数组首元素的指针, 但 decltype 返回的仍是数组.

指针作为迭代器: string arr[] = "abcde"; string* p = arr; //p 指向'a' ++p; //p 指向'b'
尾后指针 string* e = &arr[5];

标准库函数用于数组: *pbeg = begin(arr); *pend = end(arr);

数组的下标可为负, 但 vector 和 string 不行, 如: int* p = &arr[2]; //p[1]指向 arr[3], p[-2]指向 arr[0]

C 风格字符串 C-style character string: 字符串存放在字符数组 char[]中并以空字符('\0')结束.
strlen(p), strcmp(p1, p2), strcat(p1, p2), strcpy(p1, p2) //C 标准库 string 函数, 传入的指针必须指向以空字符作为结束的数组

使用 string 比 C 风格字符串更安全高效.

要使用范围 for 语句处理多维数组, 除了最内层的循环外, 其他循环的控制变量都应是引用类型, 为了避免数组被转换成指针, 如: for (auto& row : mul_arr) {for (auto col : row) {cout << col << endl;}}

参与取余%运算的对象必须是整型.

除非必须, 否则不使用后置递增递减运算 i++, i—

*p++ 等价于 *(p++): p 值加 1, 然后返回 p 的初始值副本

左移<<: 在右侧插入 n 个二进制的 0 ($\times 2^n$). 右移>>: 在左侧插入 n 个二进制的 0 ($\div 2^n$).
位求反~: 逐位求反. 位与&: 都是 1. 位或|: 至少有一个 1. 异或^: 有且只有一个 1, 否则为 0.

sizeof 运算符: 返回所占的字节数, 是 size_t 类型.

数组中元素个数: constexpr size_t sz = sizeof(arr) / sizeof (*arr);

无符号类型和带符号类型运算, 若无符号类型不小于带符号类型, 则带符号的运算对象转换成无符号的. 若无符号类型小于带符号类型, 则转换结果依赖于机器.

定义在 while 条件部分或循环体内的变量每次迭代都经历从创建到销毁的过程.

for 语句头可省略掉 init-statement, condition 和 expression 中的任意一条, 用空语句(;)代替.

break 语句负责终止离它最近的 while, do while, for 或 switch 语句, 并从这些语句之后的第一条语句开始继续执行. 注意不能终止 if 语句.

continue 语句终止最近的循环中的当前迭代并立即开始下一次迭代. 只能出现在 while, do while 或 for 循环的内部或嵌套在循环里的语句的内部.

函数的调用完成两项工作: 用实参 argument 初始化函数对应的形参 formal parameter, 将控制权转移给被调函数 called function, 此时主调函数 calling function 的执行暂时中断. 遇到 return 语句时函数结束执行过程. return 语句也完成两项工作: 返回 return 语句中的值, 将控制权转移回主调函数.

形参列表中的形参用逗号隔开，即使多个形参类型一样也必须都写出来，如: int func(int a, int b) {}

自动对象 automatic object: 只存在于块执行期间的对象. 对于普通局部变量对应的对象来说，当函数的控制路径经过变量定义语句时创建该对象，当到达定义所在块末尾时销毁它.

局部静态对象 local static object: 在执行路径第一次经过对象定义语句时初始化，直到程序终止才被销毁.

建议使用引用类型的形参代替指针.

如果函数无须改变引用形参的值，建议将其声明为常量引用 const int& a
不能把 const 对象，字面值或需要类型转换的对象传递给普通的引用形参.

数组的两个特殊性质对定义和使用作用在数组上的函数有影响: 不允许拷贝数组(所以无法值传递)和使用数组时会转换成指针. void func(const int*)等价 void func(const int[])等价 void func(const int[6])

数组引用形参: func(int (&arr) [10]) //正确, arr 是具有 10 个整数的数组的引用 func(int& arr[10]) //错误，将 arr 声明成了 10 个引用的数组

无返回值的 return 语句只能用在返回类型为 void 的函数中，其中 return 可以省略.
其中 main()例外，若 main 函数结尾没有 return，编译器隐式插入一条 return 0;语句.

不要返回局部对象的引用或指针.

递归函数 recursive function: 调用自身的函数. 在该函数中一定有某条路径是不包含递归调用的，如: int factorial(int val) {if (val > 1) {return factorial(val-1) * val;} return 1;}

函数不能返回一个数组(无法拷贝)，但可以返回数组的引用或指针，如: int (*func(int i)) [10];
或使用类型别名: using arrT = int[10]; //arrT 表示含有 10 个整数的数组的类型的别名 arrT* func(int i); //func 返回一个指向含有 10 个整数的数组的指针

int* p1[10]; int (*p2) [10] = &arr; //p1 是一个含有 10 个指针的数组, p2 是一个指向含有 10 个整数的数组的指针

尾置返回类型 trailing return type: 简化上述 func 声明. 它跟在形参列表后并以->开头，如:
auto func(int i) -> int(*) [10];

或使用 decltype，如: int arr[] = {1, 2, 3}; decltype(arr) *arrPtr(int i) {} //返回一个指向含有 3 个整数的数组的指针，注意 decltype 并不负责把数组类型转换成对应的指针，故要加一个*

main 函数不能重载.

在局部作用域中声明函数不是一个好的选择. 一般放在头文件中.

内联函数 inline 可以避免函数调用的开销(调用前要先保存寄存器，并在返回时恢复；可能需要拷贝实参；程序转向一个新的位置继续进行.) 在函数返回类型前加上关键字 inline 即可. 用于优化规模较小，流程直接，频繁调用的函数.

函数指针: 指向的是函数而非对象. 用指针替换函数名即可，如: int (*pf) (int a, int b) {}

括号必不可少, int* pf(int a, int b) {}是一个返回值为 int 指针的函数.
把函数名作为一个值使用时, 该函数自动转换为指针, 如: int func(int a, int b) {} pf = func; //pf 指向 func 函数 (等价于 pf = &func; //取址符可选)
可直接使用指针调用函数, 无须提前解引用指针: int a1 = pf(1, 2); int a2 = (*pf) (1, 2); int a2 = func(1, 2); 三者等价

把函数作为实参使用, 此时会自动转换成指针.

面向对象特性: 封装 encapsulation, 继承 inheritance, 多态 polymorphism

定义在类内部的函数是隐式的 inline 函数.

this 是一个常量指针, 指向当前对象. 成员函数通过 this 来访问调用它的对象.

常量对象, 常量对象的引用或指针都只能调用常量成员函数.

构造函数 construtor: 每个类分别定义了它的对象被初始化的方式, 类通过一个或几个特殊的成员函数来控制其对象的初始化过程.

构造函数名和类名相同, 没有返回类型, 不同的构造函数之间必须在参数数量或类型上有所区别.

只有当类没有声明任何构造函数时, 编译器才会自动地生成默认构造函数, 也称合成的默认构造函数 synthesized default constructor.

如果类包含有内置类型或复合类型的成员, 则只有当这些成员全都被赋予了类内的初始值时, 这个类才适合于使用合成的默认构造函数.

建议使用构造函数的初始值而非先定义再赋值, 如: Person(int age, string name) : m_age(age), m_name(name) {} //构造函数初始值列表. 若成员是 const 或引用时, 必须用此方法来被初始化. 最好令构造函数初始值顺序与成员声明的顺序保持一致.

如果定义了其他构造函数, 最好也提供一个默认构造函数.

访问说明符 access specifiers 加强类的封装性:
public – 成员可在整个程序内被访问.
protected – 成员可被子类对象访问, 不能在类外被访问.
private – 成员可被类的成员函数和友元访问, 不能被子类对象访问. 其部分封装了类的实现细节.

class 和 struct 定义类的唯一区别就是默认访问权限. class 是 private, 而 struct 是 public.

友元 friend: 令其他类或函数能访问类的非公有成员. 即在类中加一条以 friend 开始的声明语句. 最好在类定义开始或结束前的位置集中声明友元. 友元关系不具有传递性.

一个 const 成员函数如果以引用的形式返回*this, 那么它的返回类型是常量引用.

类的静态成员: 静态成员与类本身直接相关, 而不是与类的各个对象保持关联. 通过在成员的声明之前加上 static. 静态成员可以是 public 或 private. 可以是常量, 引用, 指针, 类等. 类似于全局变量.

类的静态成员存在于任何对象之外，可被所有对象共享. 对象中不包含与静态成员有关的数据.

类的静态成员函数也不与任何对象绑定在一起，不包含 this 指针，不能被声明成 const.

访问静态成员: Base::static_mem(); 或 Base b1; b1.static_mem();

在类外部定义静态成员时，不能重复 static. 该关键字只出现在类内部的声明语句中. 它们不是由类的构造函数初始化的. 通常情况下，类的静态成员不应该在类的内部初始化.

静态数据成员可以是不完全类型，可以作为默认实参. 而普通成员则不行.

数组的空间效率低，因为经常有空闲的内存区域. 时间效率高，读/写/查找为 *O(1)*，因为内存连续. vector 是一种动态数组，创建时先开辟小空间，当超过容量时分配更大的空间，每次扩充为 2 倍容量.

当数组作为函数参数来传递时，数组会自动退化为同类型的指针.

当几个指针赋值给相同的常量字符串时，它们实际上会指向相同的内存地址.
但把字符串赋值给数组时，指向的内存地址则不同. e.g. char str1[] = "hello world"; char str2[] = "hello world"; char* str3 = "hello world"; char* str4 = "hello world"; //str1 != str2, str3 == str4

递归本质上是一个栈结构.

C++程序在执行时，将内存大方向划分为 4 个区域.
代码区: 存放函数体的二进制代码，由操作系统进行管理.
全局区: 存放全局变量和静态变量以及常量.
栈区: 由编译器自动分配释放，存放函数的参数值和局部变量等.
堆区: 由程序员分配和释放，若程序员不释放，程序结束时由操作系统回收.
(*new* 开辟的空间在堆上，而一般声明的变量存放在栈上)

内存四区意义: 不同区域存放的数据，赋予不同的生命周期，使得编程更加灵活.

堆栈空间分配区别.
1. 栈(操作系统): 由操作系统自动分配释放，存放函数的参数值，局部变量的值等. 其操作方式类似于数据结构中的栈.
2. 堆(操作系统): 一般由程序员分配释放，若程序员不释放，程序结束时可能由 OS 回收，分配方式类似于链表.

堆栈缓存方式区别.
1. 栈使用的是一级缓存，通常都是被调用时处于存储空间中，调用完毕立即释放.
2. 堆是存放在二级缓存中，生命周期由虚拟机的垃圾回收算法来决定(并不是一旦成为孤儿对象就能被回收). 所以调用这些对象的速度要相对来得低一些.

堆栈数据结构区别.
1. 堆(数据结构): 堆可以被看成是一棵树，如堆排序(完全二叉树).
2. 栈(数据结构): 一种先进后出的数据结构.

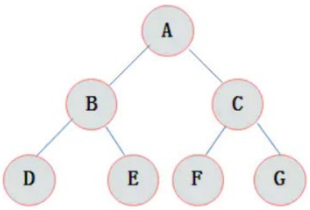
set/multiset/map/multimap 基于红黑树 RB-Tree. unordered_set/unordered_map 基于哈希表 Hash table.

树(Tree)是 n (n >= 0)个结点的有限集. n = 0 时称为空树. 在任意一颗非空树中:

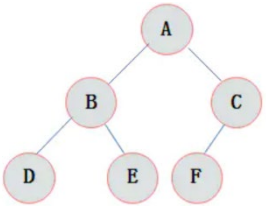
1. 有且仅有一个特定的称为根(Root)的结点.
2. 当 n > 1 时, 其余结点可分为 m (m > 0)个互不相交的有限集 T1, T2, … Tn, 其中每一个集合本身又是一棵树, 并且称为根的子树.
3. n > 0 时根结点是唯一的, 不可能存在多个根结点, 数据结构中的树只能有一个根结点.
4. m > 0 时, 子树的个数没有限制, 但它们一定是互不相交的.
5. 结点拥有的子树数目称为结点的度(Degree). 二叉树的结点的度不大于 2.

二叉树(Binary Tree)性质:

1. 左子树和右子树是有顺序的, 次序不能任意颠倒.
2. 即使树中某结点只有一棵子树, 也要区分它是左子树还是右子树.
3. 在二叉树的第 i 层上最多有 2^{i-1} 个节点. (i >= 1)
4. 二叉树中如果深度为 k, 那么最多有 $2^k - 1$ 个节点. (k >= 1)
5. $n_0 = n_2 + 1$. n_0 表示度数为 0 的节点数, n_2 表示度数为 2 的节点数.
6. 满二叉树: 所有分支结点都存在左子树和右子树, 并且所有叶子都在同一层上. 特点有:
   6.1 叶子只能出现在最下一层.
   6.2 非叶子结点的度一定是 2.
   6.3 在同样深度的二叉树中, 满二叉树的结点个数最多, 叶子数最多.



7. 完全二叉树: 对一棵具有 n 个结点的完全二叉树按层编号, 编号为 i (1 <= i <= n)的结点与同样深度的满二叉树中编号为 i 的结点在二叉树中位置完全相同. 特点有:
   7.1 叶子结点只能出现在最下层和次下层.
   7.2 最下层的叶子结点集中在树的左部.
   7.3 倒数第二层若存在叶子结点, 一定在右部连续位置.
   7.4 如果结点度为 1, 则该结点只有左子树, 没有右子树.
   7.5 同样结点数目的二叉树, 完全二叉树深度最小.
   7.6 满二叉树一定是完全二叉树, 但反过来不一定成立.



8. 具有 n 个节点的完全二叉树的深度为$[\log_2 n] + 1$, 其中$[\log_2 n]$是向下取整.
9. 若对含 n 个结点的完全二叉树从上到下且从左至右进行 1 至 n 的编号, 则对完全二叉树中任意一个编号为 i 的结点有如下特性:
   9.1 若 i = 1, 则该结点是二叉树的根, 否则, 编号为[i / 2]的结点为其双亲结点.
   9.2 若 2i > n, 则该结点无左子结点, 否则, 编号为 2i 的结点为其左子结点.
   9.3 若 2i + 1 > n, 则该结点无右子结点, 否则, 编号为 2i + 1 的结点为其右子结点.
   9.4 叶子结点总数为(n + 1) / 2.
10. 二叉树的访问次序可以分为四种. 前序遍历(preprder traversal): 根-左-右. 中序遍历(inorder traversal): 左-根-右. 后序遍历(postorder traversal): 左-右-根. 层序遍历: 按照树的层次自上而下遍历.
11. 二叉树的重建, 只能在提供前序+中序, 或者后序+中序的情况下, 才可以正常的重建.

二叉搜索树/二叉排序树/二叉查找树(Binary Sort Tree), 是一棵空树, 或具有下列性质:

1. 若左子树不空, 则左子树上所有结点的值均小于它的根结点的值.
2. 若右子树不空, 则右子树上所有结点的值均大于它的根结点的值.
3. 左, 右子树也分别为二叉排序树.
4. 没有键值相等的结点.

红黑树(Red Black Tree)

很多算法可用递归和循环两种方式实现. 通常基于递归的实现方法代码比较简洁, 但性能不如基于循环的实现方法. 因为递归是函数调用自身, 有时间和空间的消耗. 每一次函数调用都需要在内存栈中分配空间以保存参数, 返回地址及临时变量, 而且向栈里压入数据和弹出数据都需要时间. 另外递归中很多计算都是重复的. 除效率之外, 递归还可能引起调用栈溢出的问题.

冒泡排序: 从第一个数据开始, 依次比较相邻元素的大小. 如果前者大于后者, 则进行交换操作, 把大的元素往后交换. 通过多轮迭代(每轮排除最后一个元素)直到没有交换操作为止. 最好时间复杂度是 O(n), 最坏时间复杂度是 O(n*n), 平均时间复杂度是 O(n*n). 空间复杂度 O(1). 稳定性: 稳定.

插入排序: 选取未排序的元素, 插入到已排序区间的合适位置, 直到未排序区间为空. 最好时间复杂度是 O(n), 最坏时间复杂度是 O(n*n), 平均时间复杂度是 O(n*n). 空间复杂度 O(1). 稳定性: 稳定. 与冒泡排序的相同点为两者的平均时间复杂度都是 O(n*n), 且都是稳定的排序算法, 都属于原地排序. 差异点为冒泡排序每轮的交换操作是动态的, 所以需要三个赋值操作才能完成. 而插入排序每轮的交换动作会固定待插入的数据, 因此只需要一步赋值操作.

归并排序: 原理是分治法. 首先将数组不断地二分, 直到最后每个部分只包含 1 个数据. 然后再对每个部分分别进行排序, 最后将排序好的相邻的两部分合并在一起. 它的执行频次与输入序列无关, 因此时间复杂度都为 O(nlogn). 空间复杂度 O(n). 稳定性: 稳定.

快速排序: 原理是分治法. 它的每轮迭代, 会选取数组中任意一个数据作为分区点, 将小于它的元素放在它的左侧, 大于它的放在它的右侧. 再利用分治思想, 继续分别对左右两侧进行同样的操作, 直至每个区间缩小为 1, 完成排序. 最好时间复杂度是 O(nlogn), 最坏时间复杂度是 O(n*n), 平均时间复杂度是 O(nlogn). 空间复杂度 O(1). 稳定性: 不稳定.

排序算法稳定性的简单形式化定义为: 如果 $A_i = A_j$, 排序前 A_i 在 A_j 之前, 排序后 A_i 还在 A_j 之前, 则称这种排序算法是稳定的. 通俗地讲就是保证排序前后两个相等的数的相对顺序不变.

动态规划(dynamic programming): 如果求一个问题的最优解(如最大值或最小值), 且该问题可以分解成若干子问题, 且子问题之间还有重叠的更小的子问题, 可以考虑用动态规划思想. 从上往下分析问题, 从下往上求解问题. 总是从解决最小问题开始, 并把已解决的子问题的最优解储存下来(如数组中), 并组合以逐步解决大问题.

贪婪算法(greedy algorithm): 每一步都可以做出一个贪婪的选择.

位运算

位与&: 两个位都为 1 时, 结果为 1　　位或 |: 两个位都为 0 时, 结果才为 0

位非~: 0 变 1, 1 变 0　　位异或^: 两个位相同时为 0, 相异为 1

左移<<: 高位丢弃, 低位补 0　　右移>>: 正数高位补 0, 负数高位补 1, 低位丢弃

左移 n 位相当于乘以 2^n, 右移 n 位相当于除以 2^n. 因为是基于二进制运算, 所以效率比算术运算高.

正数边界值: 1~0x7FFFFFFF, 负数边界值: 0x80000000~0xFFFFFFFF.

把一个整数减 1 后再和原数做位与运算, 相当于把二进制表示的整数最右边的 1 变成 0.

判断数的奇偶(包括负数), 可和 1 做位与运算, 等于 1 为奇, 0 为偶.

由于精度原因，不能用等号判断两个小数是否相等. e.g. double d1, d2; if (d1 == d2) //错误
可以定义一个 equal()函数，如果两数之差的绝对值很小，如小于 0.0000001，就可认为它们相等.

测试用例：功能测试，边界测试，负面测试，效率测试

大数的表达：用字符串或数组来表示.

若链表中的头节点可能被删除，传入函数中的值应是**pHead 而不是*pHead.

提高代码的鲁棒性的有效途径是进行防御性编程，即预见在什么地方可能会出现问题，并为这些可能出现的问题制定处理方式. 最简单的方式是在函数开头添加代码验证输入是否符合要求，如空指针，空字符串等.

exit() 直接退出程序. exit(0)表示程序正常退出. 除 0 之外，其他参数均代表程序异常退出，e.g. exit(1), exit(-1). 而 return 表示跳出函数.

堆(Heap)是一棵完全二叉树. 堆中的某个结点的值总是大于等于（最大堆）或小于等于（最小堆）其子结点的值. 堆中每个结点的子树都是堆树.
一般用数组来表示堆. 根结点编号为 0 时，i 结点的父结点为(i – 1) / 2，左子结点为 2 * i + 1，右子结点为 2 * i + 2.
叶子结点个数(n + 1) / 2，最后一个非叶子结点为 n / 2 – 1.
堆中插入元素：新元素被加入到最后，然后更新堆以恢复排序.
堆中删除元素：删除总是发生在根结点 A[0]处. 然后用最后一个元素填补空缺，再更新堆以恢复排序.
堆排序 Heapsort 时间复杂度 O(nlogn)，空间复杂度 O(1).