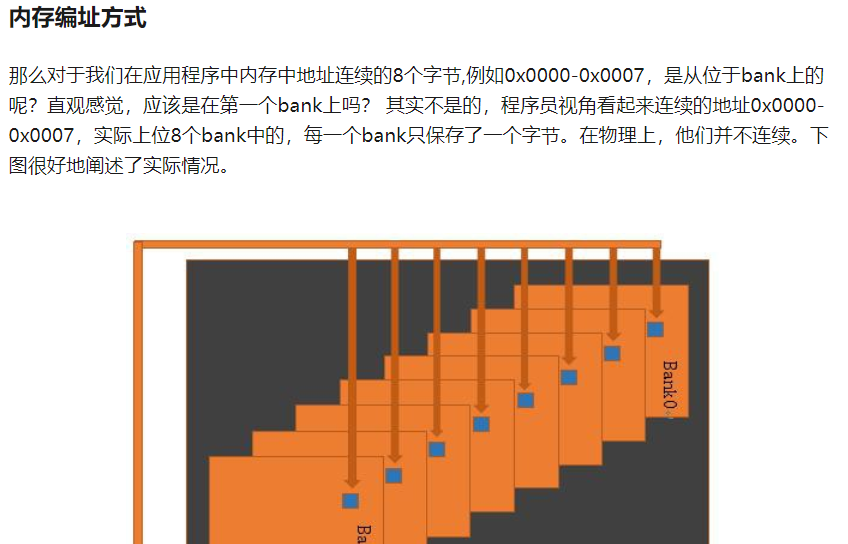
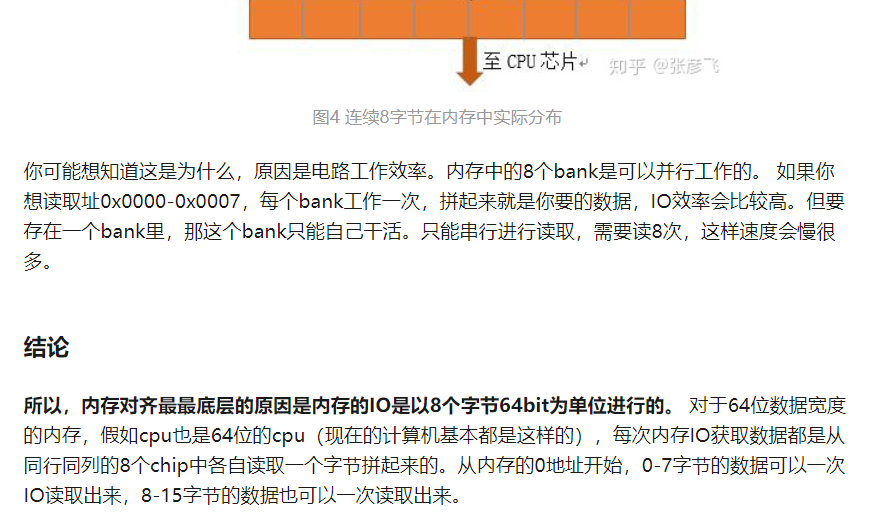
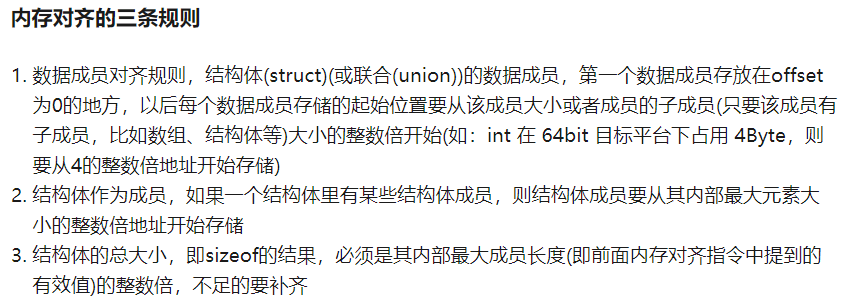
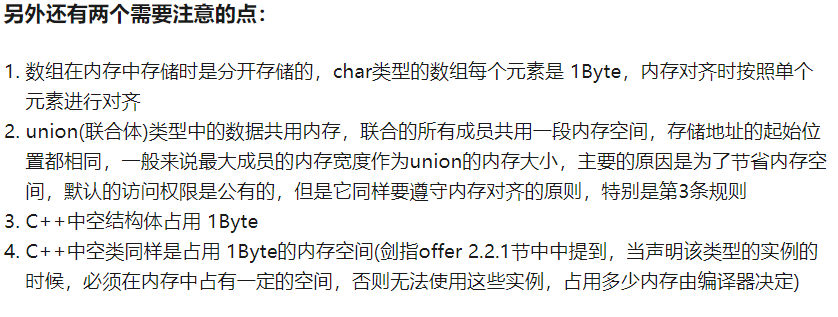
1. 数组声明及初始化
   1. int a[2];

a[0]=3;

a[1]=4;

* 1. int a[2] = { 3, 4 };
  2. 不允许将一个数组赋值给另一个数组
  3. 不允许 int a[2]; a[2] = { 3, 4 };，花括号的必须在声明时赋值
  4. 可以不声明size，通过{}中的个数编译器自己算 int a[] = {3，4}；
  5. 计算数组容量：sizeof(a) / sizeof(int)
  6. 动态声明：
     1. int\* arr = new int[10]; arr指针表示数组首位地址

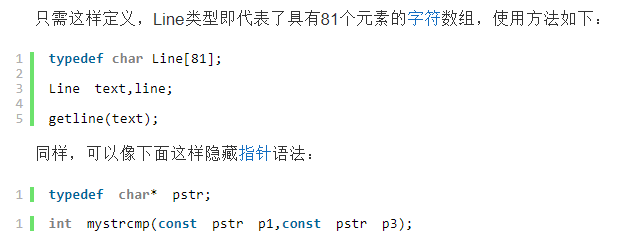
1. 字符串
   1. ‘\0’结尾的char数组
   2. 可以将一个字符串赋值给另一个
   3. 可以通过运算符+连接字符串
2. 结构体
   1. 也可以通过{}初始化，但是需要按顺序
   2. 也可以在方法内声明
3. 共用体（union）：
   1. 可以包含很多类型，但是当一种类型被赋值后，其他类型会失效
   2. 应用：比如一个物品id有id也有名字，但是想用一个变量表示
4. 指针：
   1. 大小和系统位数有关，32位4字节，64位8字节
   2. delete只能用于new创建的, 因为其实delete是为了清理堆中的内存，而不用new就不在堆中有内存
   3. 不能delete两次
   4. new[]分配的内存要delete[]
   5. 对空指针delete是安全的
5. 内存对齐 <https://zhuanlan.zhihu.com/p/93822540>
   1. 原因：<https://zhuanlan.zhihu.com/p/83449008>
   2. 
   3. 
   4. 
   5. 
6. 函数指针：
   1. 函数去掉括号即可
   2. 声明函数指针：

假设函数原型如下

double pam(int);

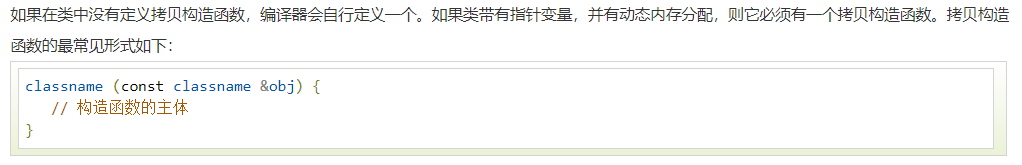
double (\*pf)(int); 即用（\*pf）替换函数名，pf即为函数指针

* 1. 调用是：(\*pf)(5), 实际上也可以这么写pf(5)，但是第一种显示的说明了正在使用函数指针, 具体为什么第二种也可以的话，暂时没有结论

1. typedef
   1. 用法1：typedef double real; 以后的double可以用Real代替
   2. 
2. const限定参数只适用于指针，用基础类型是没有意义的，因为基础类型本身就是副本，而修饰指针则表示指针指向的内存无法修改
3. 内联函数(inline)
   1. 主要为了提高运行速度
   2. 常规函数调用时，标记起始点，使程序跳到函数所在地址，并在函数结束时返回，来回跳会有开销
   3. 内联函数用相应的代码替换函数调用，用内存换时间，程序无需跳到另一个位置执行代码
   4. 但是，比如10个不同的地方调用同一个内联函数，则拷贝10份
4. 引用：
   1. 必须在声明时初始化，不能像指针先声明再赋值
   2. 引用更接近const指针，必须在创建时初始化，一旦与某个变量关联起来，将一直效忠于他，即：

int& rodents = rats; 实际上等于 int\* const pr = &rats;

* 1. 给引用赋值并不会修改引用的指向，比如上述改成rodents=x；，则rats和rodents的值都将被修改为x，但是还是原内存地址，也就是说只是将变量x赋值给了rats
  2. 有个比较复杂的事，但是有必要说清楚
     1. 当参数为引用类型时，实参和形参不匹配，则会报错
     2. 但是如果修饰引用参数为const，并且实参可以转换为形参的话，程序就会生成一个临时引用变量，用于计算，但是这种情况就类似按值传递了
     3. 至于为什么这么做呢，早期的C++，如果形参是int&，实参是Long，则不会报错，而是生成临时int变量，这样的话其实传进来的实参是不会在方法中被修改的
     4. 而新版c++默认不允许不同类型传递，除非加const
  3. &&可以修饰右值
  4. 函数返回引用效率高，一般返回值，比如struct a = change(struct x);，会将change结果(假设是个struct)复制到一个临时位置，然后然后被赋值给a，但是如果返回的是引用的话，直接返回&x，就可以直接把x地址赋值给a
  5. 拷贝构造函数，即构造函数中的参数是同类型引用，这样在用同类型赋值时就会调用拷贝构造函数，详细如下



* 1. 使用引用参数的原因：
     1. 修改此参数
     2. 传递引用而不是复制对象，提交速度
  2. **指针和引用怎么看使用什么**：
     1. 对于使用传递的值而不作修改的函数
        1. 如果数据对象很小，如内置数据类型或小型结构，按值传递
        2. 数组只能用指针，并将指针声明为const
        3. 较大结构，则使用const引用或const指针
        4. 类，使用const引用
     2. 对于修改调用函数中数据的函数：
        1. 内置数据类型使用指针，因为如果看到fixit(&x)其中x是int，就会很明显的知道此函数会修改x
        2. 数组只能使用指针
        3. 结构则使用引用或指针
        4. 类使用引用
  3. 给引用赋值其他变量只会把值付给引用，并不会转移引用
  4. 引用类型初始值必须为对象，不能int&a = 1;这样是错的

1. 代码执行内存：
   1. 文字常量区：存放常量，比如字符串和数值，并且会复用，比如int a = 1和int b = 1内存是同一个
   2. 栈区：编译器自动分配和释放，比如函数参数、局部变量等
   3. 全局区（静态区）：存放全局变量和静态变量，初始化的在一块区域，未初始化的在相邻的另一块区域
   4. 代码区：存放静态和非静态函数的二进制代码，注意代码区在程序运行的时候就都会放到内存中了
   5. 堆区：程序员手动分配的，除了上面的4个都在堆里了
2. 模板：
   1. template<typename, AnyType>
   2. 显示具体化，比如对于某个类型不想像其他类型一样的操作，可以特例
      1. template<> void swap<job>(job&, job&)
      2. 因为非模板优先于模板，显示具体化模板优先于模板，所以当实参为job类型时，会优先调用具体化模板
   3. 比如泛型相加，无法确定当前相加后的类型，可以用c++11的decltype(x+y)来作为类型,如果是decltype((x))则表示x的引用
   4. 而对于返回值的类型不确定，不能像上面那么写，因为x y还没有定义，可以这么写

template<class T1, class T2>

auto gt(T1 x, T2 y) -> decltype(x + y)

{ return x + y}

1. 反汇编看了一下，一般值传递是mov，引用传递都是lea，因为Lea不解引用,c++中不传指针和引用都是按值传递
2. const
   1. const int a = 1; const int& b = a; const的引用，不能修改b的值
   2. const int a = 1; const int\* b = &a; 指向常量的指针，即\*b=3这种操作是不允许的，因为指向的是常量，不能改变值
   3. int a = 1; int\* const b = &a; 常量指针，即int c = 2; b=&c;是不允许的，因为是常量指针，但是可以\*b=3这种可以，因为指向的不是常量
   4. const放在方法后表示此方法不可以修改类的成员变量
   5. const放在方法返回值前表示返回值不可被修改
3. explicit
   1. 修饰构造函数，表明该构造函数是显示的，由于给类赋值默认会有转换，如果类型对不上就会寻找构造函数，而加了这个关键字可以屏蔽这种隐式转换
   2. 只有一个参数的可以用，或者一个参数后面都是默认参数也可以
   3. 比如一个类构造函数参数一个是int， A a = 1 会被默认转换为int参数的构造函数，为了防止这种情况，所以用explicit修饰
   4. https://zhuanlan.zhihu.com/p/92262481
4. 静态库：.a .lib
5. 动态库：.so .dll
6. sizeof
   1. 修饰基本数据类型，返回的对应长度，
   2. 修饰指针返回的是指针长度,比如32位系统中是4字节
   3. 修饰函数，返回的是此函数返回值的长度，对void函数调用会报错
   4. 修饰数组，返回的是数组中元素数量\*元素所占长度
   5. 修饰结构体或类，返回的是对应总长度，注意这里涉及到内存对齐
   6. 修饰引用，被引用对象的长度