C/C++

* C语言补充
  + printf
    - 进制
      * d
      * o
      * x
        + 大小写
        + 加#号
  + switch
    - 支持byte，char，short，int，long，bool，整数类型和枚举
    - 不支持float，double，string
    - case里为何不能定义变量
  + 动态内存管理
    - malloc函数
      * 函数原型：void \*malloc(size\_t size);
      * 分配成功则返回指向被分配内存的指针（内存中的值未初始化）分配失败返回NULL
      * 在 1G 内存的计算机中能否 malloc(1.2G) ？为什么？
        + 能，malloc能够申请的空间大小与物理内存的大小没有直接关系，仅与程序的虚拟地址空间相关
      * 底层原理
        + 为了减少内存碎片和系统调用的开销，malloc采用内存池的方式

先申请大块内存作为堆区然后将堆区分为多个内存块，以块作为内存管理的基本单位。当用户申请内存时，直接从堆区分配一块合适的空闲块

malloc采用隐式链表结构将堆区分成连续的、大小不一的块，包含已分配块和未分配块

malloc采用显示链表结构来管理所有的空闲块：即用一个双向链表将空闲块连接起来，每个空闲块记录了一个连续、未分配的地址

当进行内存分配时，malloc会通过隐式链表遍历所有的空闲块，选择满足要求的块进行分配；当进行内存合并时，malloc采用边界标记法，根据每个块的前后块是否以及分配来决定是否进行块合并

* + - * + 申请内存时，一般会通过brk或者mmap系统调用进行申请

当申请内存小于128k时，会使用系统brk在堆区中分配

当申请内存大于128k时，会使用系统函数mmap在映射区分配

* + - realloc函数
      * 函数原型：extern void \*realloc(void\* mem\_address, unsigned int newsize);指针名 = （数据类型 \*）realloc（要改变内存大小的指针名，新的大小）
      * 假如旧内存后面还有足够多的剩余内存：realloc的内存=旧内存+剩余内存
      * 假如旧内存后面无足够多的剩余内存：realloc将申请新的内存，把旧内存中的数据拷贝到新内存中，再将原内存释放，返回新内存的地址
      * 传递给realloc的指针必须是先通过malloc,calloc或者是realloc分配的
      * 传递给realloc的指针可以为空，等同于malloc
    - calloc函数
      * 函数原型：void \*calloc(size\_t num,size\_t size);
      * 与malloc区别
        + calloc分配完内存后，会将该内存空间置零，而malloc不初始化，内存中的是随机数据
  + 字符串操作
    - strcpy
      * des 和 src 所指内存区域不可以重叠且 des 必须有足够的空间来容纳 src 的字符串
      * char \* myStrcpy(char \*des, const char\*src) { assert((des != NULL) && (src != NULL)); char \* addr = des; while ((\*des++ = \*src++) != '\0'); return addr;}
      * strcpy 会拷贝’\0’
    - strcmp
      * /\* \*函数strcmp的原型是int strcmp(const char \*s1,const char \*s2)。 若s1==s2，返回零；若s1>s2，返回正数；若s1<s2，返回负数。\*/int myStrcmp(const char\* s1, const char\* s2) { assert((s1 != NULL) && (s2 != NULL)); while (\*s1 == \*s2) { if (\*s1 == '\0') return 0; ++s1; ++s2; } return \*s1 - \*s2;}
    - strlen
      * int myStrlen(const char\* src) { assert(src != NULL); int len = 0; while ((\*src++) != '\0') { ++len; } return len;}
    - strcat
      * //des 和 src 所指内存区域不可以重叠且 des 必须有足够的空间来容纳 src 的字符串char \* myStrcat(char\*des, const char \*src) { assert((des != NULL) && (src != NULL)); char \*addr = des; while (\*des != '\0') {//移动到字符串末尾 ++des; } //此时des指向的是‘\0’ while (\*des++ = \*src++) { } return addr;}
  + 内存管理
    - memcpy
      * void \*memcpy(void \*destin, void \*source, unsigned n)；
      * 从源内存地址的起始位置开始拷贝若干个字节到目标内存地址中，即从源source中拷贝n个字节到目标destin中
      * source和destin所指的内存区域可能重叠，但是如果source和destin所指的内存区域重叠,那么这个函数并不能够确保source所在重叠区域在拷贝之前不被覆盖。而使用memmove可以用来处理重叠区域。函数返回指向destin的指针
      * 底层原理
    - memset
    - memcmp
  + 内存分配方式
    - 从静态存储区分配
      * 内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量,static变量。
    - 在栈上创建
      * 在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
    - 在堆上分配
      * 动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。动态内存的生存期由程序员决定，使用非常灵活，但如果在堆上分配了空间，就有责任回收它，否则运行的程序会出现内存泄漏，频繁地分配和释放不同大小的堆空间将会产生堆内碎块。
  + 宏定义
    - 文件包含#include
    - 宏定义#define
      * #define 标识符 字符串
        + #define PI 3.14
        + #define S(a,b) a\*b

S(2,3) ==6

* + - * + #define S(a) a\*a

S(2+3) == 2+3\*2+3 = 11

* + - * + 交换两个数

#define SWAP(a,b) (a)=(a)+(b);(b)=(a)-(b);(a)=(a)-(b);

* + - * + 找较小数

#define MIN(a,b) ((a) <= (b) ? (a) : (b) )

* + - * + 数组大小

#define LENOFARR(arr) sizeof(arr)/sizeof(arr[0])

* + - * + 不使用sizeof，求类型所占的字节数

#define SIZEOF(value) (char \*)(&value+1)-(char\*)(&value)

（char\*）(&value)返回的是value地址的第一个字节所在的地址（char\*）(&value+1)返回的是value地址的下一个地址空间的第一个字节所在的地址

int s = 1;double d = 2.0;float f = 1.0;cout << SIZEOF(s) << endl;//4cout << SIZEOF(d) << endl;//8cout << SIZEOF(f) << endl;//4

* + - * 宏名一般用大写，提高可读性和通用性，如数组的大小
      * 宏定义末尾不加分号
      * 无类型，不做安全检查
      * 不分配内存
      * 可以嵌套
      * 可使用#undef命令终止宏定义的作用域
    - 条件编译#ifdef...
      * 防止头文件被重复引用
    - #error
      * 编译程序时，只要遇到#error
  + 判断一段程序是C编译还是C++
    - #ifdef \_\_cplusplus cout << "C++" << endl;#else cout << "C" << endl;#endif
  + 在main函数之前运行的函数
    - C++
      * 全局类变量的构造函数、拷贝构造函数
      * 匿名函数
        + int fun1 = []() { cout << "fun1" << endl; return 0;}();
      * 用函数返回值给全局变量初始化
  + 数组的下标可以是负数
    - 下标只是给出了一个与当前地址的偏移量，只要根据这个偏移量能定位到目标地址即可
      * int nums[] = { 1,2,3,4,5 };int \*p = nums + 4;cout << p[-4] << endl;//1
  + 可变参数函数
    - 比如printf和scanf
    - 实现原理
      * 利用四个宏（头文件stdarg.h）
        + va\_list
        + va\_start
        + va\_arg
        + va\_end
      * 内存压栈技术
        + 压栈的顺序是从最右边参数开始的，再向左逐个压入，根据栈的原理，在取参数时，就从第一个可变参数开始
  + 函数调用
    - 每一个函数调用都会分配函数栈，在栈内进行函数指向过程。调用前，先把返回地址压栈，然后把当前函数的esp指针压栈
    - 参数压栈顺序
      * 从右至左
  + 位运算