C++5

* 编译底层
  + 内存对齐
    - 平台、移植原因：不是所有的硬件平台都能访问任意地址上的任意数据；某些硬件平台只能在某些地址处取某些特定类型的数据
    - 提升CPU对内存的访问速度
    - 规则
      * 数据成员的起始地址为该成员大小的整数倍，起始地址从0开始
      * 结构体成员按照其中最大元素的大小的整数倍地址开始
      * 结构体总大小，要为内部最大元素的整数倍
  + 内存管理
    - 堆
      * 一般由程序员负责分配和释放，若程序员不释放，程序结束时可能由OS回收
    - 栈
      * 编译器自动分配释放 ，存放函数的参数值，局部变量的值
        + 栈溢出

没有回收垃圾资源

递归的层次太深

* + - 代码区
      * 存放函数的二进制代码
    - 全局/静态存储区
      * 全局变量和静态变量被分配到同一块内存中（C语言又分为初始化和未初始化的）
    - 常量存储区
      * 存放常量，不允许修改
  + 内存碎片
    - 分类
    - 解决方案
  + 编译流程
    - 预编译（预处理）
      * 处理“#”开头的关键字，#define，#ifdef,#ifndef ，#include
      * 删除注释，添加行号和文件标识
    - 编译
      * 语义分析
      * 语法分析
      * 内联函数的替换
      * 安全性合法性检查
    - 汇编
      * 将汇编代码转化成机器可执行的命令（目标代码）
    - 链接
      * 将编写的目标代码，系统启动代码和库代码合并成可执行文件
        + 静态链接
        + 动态链接
  + include头文件双引号""和尖括号<>的区别
    - 编译器预处理阶段查找头文件的路径不一样
    - <>查找头文件优先编译器设置的头文件路径
    - ""查找头文件优先头文件目录
  + 库
    - 现有写好的、成熟的、可复用的，是一种可执行代码的二进制形式，静态动态是指链接
    - 静态库
      * 在链接阶段，会将汇编生成的目标文件.o和引用到的库链接到可执行文件中，静态库文件是.a（Linux）、.lib（Windows）文件
      * 静态库对函数库的链接是放在编译时期完成的，程序在运行时与函数库再无瓜葛，移植方便
      * 浪费空间和资源，所有相关的目标文件和函数库都被链接组合到一个可执行文件中
      * 静态库的更新会对程序的更新、部署、分布带来问题，静态库更新了，所有使用它的程序都要重新编译发布
    - 动态库
      * 动态库在运行阶段才会被载入，内存只需有一份该共享库的实例，实现进程之间的资源共享（动态库也称共享库），规避了空间浪费问题，解决了静态库的更新问题。用户只需更新动态库就绪，增量更新。
  + 重载的底层原理
    - 利用 name mangling（倾轧）技术，来改变函数名，区分参数不同的同名函数，Linux下可用nm teset.o查看
  + 虚拟内存VM
    - 虚拟内存让每个进程都有独立的地址空间，保护了每个进程的地址空间不会被其他进程破坏，一个磁盘文件对象可被多个进程共享访问
    - 虚拟化内存简化了内存管理，内存保护，内存效率高
    - 内存映射机制：初始化虚拟内存时，会把虚拟内存和磁盘空间对象对应起来
  + 大小端定义及判断
    - 大端：寄存器的低地址存放字节数据的高位
    - 小端：寄存器的低地址存放字节数据的低位
    - 判断：定义char指针指向short，输出char的地址和加一地址处的值 short i = 0x1122; char\* a = (char \*)(&i); printf("%p\n",a);//低地址 printf("%x\n",\*a);//22,低位 printf("%p\n",(a+1));//高地址 printf("%x\n",\*(a+1));//11，高位 //说明是小端
  + C++程序组成
    - 头文件
    - 类型声明，全局变量
    - 主函数
  + 编译内存相关
    - 变量的区别
    - 全局变量定义在头文件
    - 内存泄漏
      * 定义
        + 由于疏忽或错误造成了程序未能释放掉不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。
      * 分类
        + 堆内存泄漏

new/malloc和delete/free没有成对使用

* + - * + 系统资源泄漏

程序使用系统分配的资源，比如Bitmap，handle,socket等没有使用相应的函数释放掉，造成系统资源的浪费

* + - * + 多态中没有将基类的析构函数定义为虚函数

导致子类资源没有被正确释放

* + - * 如何防止
        + 使用智能指针
      * 内存泄漏检测
        + new和delete是否成对
        + 记录申请和释放的对象是否成对，在类中追加一个静态变量
        + 在Linux下可使用内存泄漏检测工具Valgrind
      * 检测原理
      * 后果
        + 占用大量内存，导致无内存可用而崩溃
        + 占用大量内存，导致其他程序无法正常使用
    - 使用new、delete常见问题
      * 忘记释放内存
      * 使用已经释放掉内存的对象
      * 同一块内存释放两次
    - 段错误
      * 段错误通常发生在访问非法内存地址的时候
        + 使用野指针
        + 试图修改字符串常量的内容