# NDK笔记

# C和C++语言

## C语言-指针入门与基本数据类型

### NDK是什么

NDK(C/C++) <-------------------> SDK(Java)性质是一样的

定义：Native Development Kit，是 Android的一个工具开发包

作用：快速开发C、 C++的动态库，并自动将so和应用一起打包成 APK

提供了把.so和.apk打包的工具

提供了交叉编译器，用于生成特定的CPU平台动态库

提供了很多很多的native支持，就相当于SDK提供了很多上层的支持一样

特点:

运行效率高

代码安全性高

功能拓展性好

易于代码复用和移植

### 说明C语言代码块结构

#include <stdio.h>

// <> 寻找系统的资源  
// “” 寻找我们自己写的资源  
// .h .hpp(声明文件 头文件)  
// .c .cpp （实现文件）  
  
// 代码结构  
int mainT1() { // 函数的主入口  
 printf("坚强的");  
  
 getchar(); // 阻塞程序  
 return 0;  
}

### 02-基本数据类型占位

#include <stdio.h>

// 基本数据类型  
int mainT2() {  
  
 int i = 100;  
 double d = 200;  
 float f = 200;  
 long l = 100;  
 short s = 100;  
 char c = 'd';  
 // 字符串  
 char \* str = "Derry";  
  
 // 不是随便打印的，需要占位  
 printf("i的值是：%d\n", i); // d == 整形  
 printf("d的值是:%lf\n", d); // lf == long float  
 printf("f的值是:%f\n", f); // f == float  
 printf("l的值是:%d\n", l); // d == 整形  
 printf("s的值是:%d\n", s); // s == short  
 printf("c的值是:%c\n", c); // c == char  
 printf("字符串:%s\n", str); // s == String  
  
 printf("Derry");  
 // getchar();  
}

### 03-基本类型占用的字节数

#include <stdio.h>

// 基本类型占用的字节数(后续过程中，非常常用的) sizeof获取字节数  
int mainT3() {  
  
 // int 占几个字节 == 4  
 // double 占几个字节 == 8  
 // char 占几个字节 == 8  
  
 printf("int 数据类型所占的字节数：%d\n", sizeof(int));// 4  
 printf("double 数据类型所占的字节数：%d\n", sizeof(double));// 8  
 printf("char 数据类型所占的字节数：%d\n", sizeof(char));// 1  
  
 return NULL; // NULL == 0  
}

04-初次感受，万物皆地址

#include <stdio.h> // C99 标准库

// 地址概念  
// Java 万物皆对象  
// C C++(对象) 万物皆指针  
// Linux 万物皆文件  
int mainT4() {  
  
 // 指针 == 地址  
  
 // %p == 地址输出的占位  
 // & == 取出地址  
  
 int number1 = 100000;  
 printf("此number1变量的地址是:%p\n", &number1); // 地址 == 00B3FE90  
  
 return NULL;  
}

#include <stdio.h>

// 指针难度高一点  
int mainT5() {  
  
 int number\_int = 100;  
 double number\_double = 200;  
 printf("number\_int的值是:%d\n", number\_int);  
 printf("number\_double的值是:%lf\n", number\_double);  
  
 // \* == 取出number\_int地址所对应的值 == 100  
 // 指针取出值  
 // 既然任何变量都是地址，可以使用都在获取值（灵感）  
 printf("number\_int的值是:%d\n", \*(&number\_int));  
 printf("number\_double的值是:%lf\n", \*(&number\_double));  
  
  
 // ================ 难度  
  
 int \* intP = &number\_int;  
 double \* doubleP = &number\_double;  
  
 // 输出 100 和 200  
 printf("number\_int的值是:%d\n", \*intP);  
 printf("number\_double的值是:%lf\n", \*doubleP);  
  
 // 概念  
 /\*\*  
 同学们：如果你去上网查询，指针的文章，就会发现，一些专业术语，导致大学老师一直说不清楚，成功劝退很多学生放弃：  
 内存地址变量，内存地址别名，指针，指针变量，指针别名，.... 一大段的专业术语 ...  
  
  
 理解： 大道至简 （化繁为简）  
  
 内存地址 == 指针  
 指针 == 内存地址  
 int \*， double \* (int类型的指针)  
  
 指针别名，指针变量 == 就是一个变量而已，只不过 是指针的 变量 而已  
 intP， doubleP  
 \*/  
  
 return NULL;  
}

#include <stdio.h>

// 通过 指针（地址） 修改值  
int mainT6() {  
  
 int i = 100;  
  
 // p:我只接收地址，你给我一个100，不规范  
 // int \* p = i;  
 int \* p = &i;  
  
 i = 200;  
  
 printf("i的值是:%d\n", i);  
  
 \*p = 300;  
  
 printf("i的值是:%d\n", i);  
  
 return 0;  
}

#include <stdio.h>

// C语言不允许函数重载， Java可以，C++可以  
  
// void change(int i); // 先声明  
  
void change(int \* i); // 先声明  
  
// 函数 取修改一个数（指针操作）  
int mainT7() {  
  
 int i = 100;  
  
 change(&i);  
  
 printf("%d\n", i);  
  
 // \* i 和 \*i 有什么区别呢 没区别 写法而已  
  
 /\*  
 int \* i; 定义指针  
  
 \*i = 888; 取出内存地址所对应的值修改成 888  
 \*/  
  
 return 0;  
}  
  
// 函数不能写在 main的下面，会报错  
// void change(int i) {}  
  
// 再实现  
/\*void change(int i) {  
  
 printf("change内存地址：%p\n", &i);  
  
 i = 200;  
}\*/  
  
// 再实现  
// 使用指针 来修改  
void change(int \* i) {  
  
 \*i = 666;  
}

#include <stdio.h>

// 由于C语言不支持函数重载，所以声明的函数，不需要写参数  
  
void changeAction();  
  
int main() {  
 int a = 100;  
 int b = 200;  
  
 changeAction(&a, &b);  
  
 printf("交换完成后的效果：%d,%d\n", a, b);  
  
}  
  
// 只接收指针(内存地址) 内存地址 == 指针  
void changeAction(int \* a, int \* b) {  
  
 int temp = \*a;  
  
 \*a = \*b;  
  
 \*b = temp;  
}

## 函数指针与指针运算

### 多级指针。

#include <stdio.h>

void update(int \* i) { // 1000H内存地址  
 // printf("update函数，i的内存地址是:%p\n", &i); // 2000H  
  
 printf("update函数，i的内存地址是:%p\n", i); // 1000H  
  
 \* i = 999;  
}  
  
// 上节课同学问了我一个问题（为什么 main函数 和 update 函数 i 的内存不地址一样？）  
int mainT1() {  
 int i = 100; // 1000H的内存地址  
 update(&i);  
  
 printf("main函数，i的内存地址是:%p\n", &i);  
  
 return 0;  
}

#include <stdio.h>

// 多级指针的意义是什么啊 (C的框架，再去说，比较时候)  
void test1(int \*\* n) {}  
  
// 1.多级指针。  
int mainT2() {  
  
 int num = 999;  
  
 // 一级指针 \*  
 // 在真实开发过程中，最多三级指针 int \*\*\*  
 int \* num\_p = &num; // 取出num的内存地址给 num\_p（一级指针）  
  
 int \*\* num\_p\_p = &num\_p; // 取出num\_p的内存地址给 num\_p\_p（二级指针）  
  
 int \*\*\* num\_ppp = &num\_p\_p;  
  
 printf("num\_p的值是：%p， num\_p\_p的值是：%p\n", num\_p, num\_p\_p);  
 // num\_p的值是：0061FF18， num\_p\_p的值是：0061FF14  
  
 // 我先输出 999  
 printf("获取最终的值：%d\n", \*\*num\_p\_p);  
 printf("获取最终的值：%d\n", \*\*\* num\_ppp);  
  
 // 多级指针的意义是什么啊  
 // test1(num\_p\_p)  
  
 // printf("num\_p的自己的内存地址是：%p， num\_p\_p自己的额内存地址是：%p\n", &num\_p, &num\_p\_p);  
  
 // \*num\_ppp 取的是什么？  
  
 return 0;  
}

### 2.数组与数组指针。

#include <stdio.h>

// 数组与数组指针。  
int mainT3() {  
 // 定义数组  
 // int [] arr = {1,2,3,4}; 错误的写法  
 int arr[] = {1,2,3,4};  
  
 // 遍历数组  
 // 其他平台不能用 Clion能用， Linux上报错  
 /\*for (int i = 0; i < 4; ++i) {  
  
 }\*/  
  
 // VS非常严格，规范， CLion包容  
  
 // Linux上没有问题，Clion没有问题，Mac没有问题  
 int i = 0;  
 for (i = 0; i < 4; ++i) {  
 printf("%d\n", arr[i]); // 取值  
 }  
  
 // 数组 和 指针 挂钩  
 // 数组的内存地址 == 第一个元素的内存地址 == &arr  
 // 数组的内存地址 == 第一个元素，不是第二个元素，也不是第n个元素  
 printf("arr = %d\n", arr);  
 printf("&arr = %d\n", &arr);  
 printf("&arr[0] = %d\n", &arr[0]);  
  
 // 既然数组就是一个内存地址  
 int \* arr\_p = arr;  
  
 printf("%d\n", \*arr\_p); // \*arr\_p 取出元素一内存地址的值 1  
  
 arr\_p ++; // 指针挪动 元素二的内存地址了  
  
 printf("%d\n", \*arr\_p); // \*arr\_p 取出元素二内存地址的值 2  
  
 arr\_p += 2;  
  
 printf("%d\n", \*arr\_p); // 输出4  
  
 // 输出1  
  
 arr\_p -= 3; // 挪动指针指向到 元素一  
  
 printf("%d\n", \*arr\_p);  
  
 arr\_p += 2000;  
  
 printf("%d\n", \*arr\_p); // 系统值 572662306  
  
  
 // 以后：我想 三维数组/三级指针  
 // int \*\*\* arrPpp;  
  
 return 0;  
}

### 3.采用指针遍历数组。

#include <stdio.h>

// 3.采用指针遍历数组。  
  
int mainT4() {  
  
 // 定义数组  
 // int [] arr = {1,2,3,4}; 错误的写法  
 int arr[] = {1,2,3,4};  
  
 // 数组是连续的内存空间（没有断层，有规律） 数组 每次挪动 4个字节 == int数组  
  
 int \* arr\_p = arr;  
  
 int i = 0;  
 for (i = 0; i < 4; ++i) {  
 printf("位置%d的值是:%d\n", i, \* (arr\_p + i));  
  
 // 04 08 12 16  
 printf("位置%d的内存地址是:%p\n", i, (arr\_p + i));  
 }  
  
 return 0;  
}

### 4.循环时给数组赋值。

#include <stdio.h>

// 4.循环时给数组赋值。  
int mainT5() {  
  
 // 定义数组  
 // int [] arr = {1,2,3,4}; 错误的写法  
 int arr[4];  
  
 int \* arrP = arr;  
  
 // sizeof arr == sizeof(arr)  
  
 // 循环赋值操作  
 int j = 0;  
 for (j = 0; j < 4; ++j) {  
 // 1.拿到 元素一 元素二 元素三 元素四 的内存地址 (arrP + j)  
 // 2.取出 元素一 元素二 元素三 元素四 的内存地址 所对应的值 \* (arrP + j)  
 \* (arrP + j) = (j + 10001);  
 }  
  
 // sizeof arr = 16(sizeof arr) / 4(sizeof(int))  
 // 变量 刚刚赋值的数组  
 for (int i = 0; i < sizeof arr / sizeof(int); ++i) {  
 printf("位置%d的值是:%d\n", i, \* (arrP + i));  
 }  
  
 return 0;  
}

### 5.数组指针操作的几种方式。

#include <stdio.h>

// 5.数组指针操作的几种方式。  
int mainT6() {  
  
 int arr[] = {1,9,0,9999};  
  
 int \* arrP = arr;  
  
 for (int i = 0; i < 4; ++i) {  
 // printf("%d\n", arrP[i]);  
 // printf("%d\n", \*(arrP + i)); // + i 挪动元素3内存地址 再取元素3内存地址所对应的值  
 // printf("%d\n", \*arrP + i); 放后面解释  
 }  
  
 return 0;  
}

### 6.指针类型有何用？。

#include <stdio.h>

// 6.指针类型有何用？。  
int mainT7() {  
  
 int num = 12;  
  
 int \* num\_p = &num;  
  
 // 优化处理，报错：检测无法通过的  
 double \* num\_p\_d = num\_p;  
  
 printf("%d\n", sizeof num\_p);  
 printf("%d\n", sizeof num\_p\_d);  
  
 // 指针占用的内存大小是？ int double xxx 的指针 永远都是 4个字节（32） 4\*2（64）  
  
 // 6.指针类型有何用？。 既然都是 4个字节，为什么还要分 指针类型  
 // 答：取值的时候，怎么取，这就是类型规定的好处  
  
 // int \* p; // 类型是为了计算偏移量  
  
 char c = 'a';  
 // char \* p = &c;  
 // char \* p = "AAAA";  
  
 return 0;  
}

### 7.函数指针。

#include <stdio.h>

void add(int num1, int num2); // 先声明  
  
void mins(int num1, int num2) {  
 printf("num1 - num2 = %d\n", (num1 - num2));  
}  
  
// 操作 回调到 add mins  
// void(\*method)(int,int) 声明好 函数指针  
// void 返回值  
// (\*method) 函数名  
// (int,int) 两个参数  
void opreate(void(\*method)(int,int), int num1, int num2) {  
 method(num1, num2);  
  
 printf("opreate函数的 method指针是多少：%p\n", method);  
}  
  
// 7.函数指针。（回调) Java接口的回调  
int mainT8() { // 【第一种写法】  
  
 opreate(add, 10, 10);  
 opreate(mins, 100, 10);  
  
 // 原理是什么？  
  
 printf("main函数的 add指针是多少：%p\n", add);  
 printf("main函数的 mins指针是多少：%p\n", mins);  
  
 // &add和add是一样的值吗  
 printf("%p, %p\n", add, &add); // 004018CE, 004018CE 一样的  
  
 return 0;  
}  
  
// 再实现 使用  
void add(int num1, int num2) {  
 printf("num1 + num2 = %d\n", (num1 + num2));  
}

#include <stdio.h>

void callBackMethod(char \* fileName, int current, int total) {  
 printf("%s图片压缩的进度是：%d/%d\n", fileName, current, total);  
}  
  
// 压缩的方法  
// 定义函数指针: 返回值(\*名称)(int,double)  
void compress(char \* fileName, void(\*callBackP)(char \*,int,int)) {  
 callBackP(fileName, 5, 100); // 回调给外交 压缩的进度情况  
}  
  
// 函数指针2  
int main() {  
  
 // 1 如果有问题  
 // VS Clion 通过了， Linux 可能不通过（因为这样不合规范）  
 // void (\* call) (char \*, int ,int) = callBackMethod;  
  
 // 有种temp的思路一样的感觉  
 // 2 再换成这种方式 【第二种写法】  
 // Linux 先定义， 再赋值  
 void (\* call) (char \*, int ,int);  
 call = &callBackMethod;  
  
 void (\* call2) (char \*, int ,int);  
 void (\* call3) (char \*, int ,int);  
 void (\* call4) (char \*, int ,int);  
 void (\* call5) (char \*, int ,int);  
 call5 = &callBackMethod; // &callBackMethod 1000H == callBackMethod 1000H  
  
 compress("derry.png", call5);  
  
 // 常量指针 指针常量 【函数指针】  
  
 // 字符串 操作符重载 + C函数  
 // char \* c = "Derry" + "A";  
}

## 静态开辟内存与动态开辟内存

### 函数指针

#include <stdio.h>

void add(int i,int j) {  
 printf("i+j=%d\n", i+j);  
}  
  
void add2(int i,int j) {  
 printf("2 i+j=%d\n", i+j);  
}  
  
// 函数 函数指针声明来接收函数  
void test(void(\*p)(int, int)) {  
 p(9, 9); // 上节课 省略\*  
  
 (\*p)(9, 9);  
 (p)(9, 9); // 省略\*  
  
 // (&p)(9, 9); 在源码没有看到这种写法  
  
 // 思考：p 为什么也可以不用写 \* 函数的上面已经声明就是 函数指针，所以可以省略\*  
  
}  
  
// 上节课 点 函数指针  
int mainT1() {  
 test(add);  
 test(add2);  
 return 0;  
}

### 1.如何自己学会阅读C语言API文档。

#include <stdio.h>

#include <stdlib.h>  
#include <time.h>  
  
// 1.如何自己学会阅读C语言API文档  
/\*  
int main() {  
  
 // 看文档的思路：  
 // 1.看到文档，不要Copy，初学者  
 // 2. 传入 &t &p 可以直接传递 NULL 尝试  
  
 // 时间单位 给他才行  
 srand((unsigned) time(NULL));  
  
 int i;  
 for (i = 0; i< 10 ; i++) {  
 printf("随机数%d\n", rand() % 100);  
 }  
  
 return(0); // return 0;  
}  
\*/  
  
// 字符串的Copy工作  
/\*函数名: strcpy  
 功 能: 串拷贝  
 用 法: char \*strcpy(char \*str1, char \*str2);  
程序例:\*/  
  
#include <stdio.h>  
#include <string.h>  
  
/\*  
int mainT2(void)  
{  
 char string[10]; // 定义一个数组  
 char \*str1 = "abcdefghi"; // 字符串定义  
  
 strcpy(string, str1); // 就是把 str1 复制到 数组  
 printf("%s\n", string); // 打印数组  
 return 0;  
}\*/  
  
// C中的 布尔  
int mainT2() {  
  
 // Java Boolean  
  
 // 非0即true， 0就是false  
  
 if (0) {  
 printf("真\n");  
 } else {  
 printf("假\n");  
 }  
  
 if (43665) {  
 printf("真\n"); // 1 走  
 } else {  
 printf("假\n"); // 2  
 }  
  
 if (-545) {  
 printf("真\n"); // 1  
 } else {  
 printf("假\n"); // 2  
 }  
  
  
 return 0;  
}

### 2.静态开辟。

#include <stdio.h>

#include <unistd.h> // 小写的  
  
// 函数进栈 定义一个int arr[5]; 定义一个 int i; (静态的范畴)  
  
// 进栈  
void staticAction() {  
 int arr[5]; // 静态开辟 栈区 （栈成员）  
  
 for (int i = 0; i <5; ++i) {  
 arr[i] = i;  
 printf("%d, %p\n", \*(arr + i), arr + i);  
 }  
} // 函数的末尾会弹栈（隐士）：执行完毕会弹栈 会释放所有的栈成员  
  
// 2.静态开辟。  
int mainT3() {  
 // int 4 \* 10 = 40M  
 // int arr[10 \* 1024 \* 1024]; // 10M \* 4 = 40M 会栈溢出  
  
 // int arr[1 \* 1024 \* 1024]; 会栈溢出  
  
 int arr[(int)(0.2 \* 1024 \* 1024)]; // 不会栈溢出  
  
 // 栈区：占用内存大小 最大值： 大概 2M 大于2M会栈溢出 平台有关系的  
 // 堆区：占用内存大小 最大值： 大概80% 40M没有任何问题，基本上不用担心 堆区很大的  
 // 大概80%： Windows系统 给我们的编译器给予的空间 的 百分之百八十  
  
 while (9) {  
 sleep(100);  
 staticAction(); // 调用开辟20  
 }  
  
 return (0);  
}

### 3.动态开辟。

#include <stdio.h>

#include <stdlib.h>  
#include <unistd.h> // 小写的  
  
/// 函数进栈 定义一个int arr[5]; 定义一个 int i; (静态的范畴)  
  
// malloc 在堆区开辟的内存空间 ， (动态的范畴)  
  
// C的开发过程中，不能出现，野指针，悬空指针  
void dynamicAction() {  
 int \* p; // 野指针 没有地址的，空的  
  
 // void \* 可以任意转变 int\* double \*  
 int \* arr = malloc(1 \* 1024 \* 1024); // 堆区开辟 4M  
  
 printf("dynamicAction函数，arr自己的内存地址:%p，堆区开辟的内存地址:%p\n", &arr, arr);  
  
 // C工程师，堆区开辟的空间，必须释放  
 free(arr); // 释放掉  
 arr = NULL; // 重新指向一块内存地址00000  
  
 printf("dynamicAction函数2 堆区开辟的内存地址:%p\n", arr); // 悬空指针  
}  
  
int mainT1() {  
  
 while (9) {  
 // sleep(100);  
 dynamicAction(); // 调用开辟20  
 }  
  
 return 0;  
}

### 4.动态开辟的使用的场景。

#include <stdio.h>

#include <stdlib.h>  
  
// 动态开辟的使用的场景  
int mainT2() {  
 // 静态开辟的内存空间大小，是不能修改的，如果不需要动态修改空间大小，当然使用 栈区 【尽量使用 静态开辟的，如果实在是需要动态改变，才使用下面】  
 // int arr [6];  
  
  
 // =================================== 下面是 动态开内存的使用场景  
  
 // 开辟的空间 想要变化， 动态范畴  
  
 int num;  
 printf("请输入数的个数：");  
  
 // 获取用户输入的值  
 scanf("%d", &num);  
  
 // 动态开辟 用户输入的值 空间的大小 【堆区】  
 int \* arr = malloc(sizeof(int) \* num);  
 // int arr2 [] == int \* arr 一样的了  
  
 int print\_num;  
 // 循环接收  
 for (int i = 0; i < num; ++i) {  
 printf("请输入第%d个的值:", i);  
  
 // 获取用户输入的值  
 scanf("%d", &print\_num);  
  
 arr[i] = print\_num;  
 printf("每个元素的值:%d, 每个元素的地址:%p\n", \*(arr + i), arr + i);  
 }  
  
 // for 循环打印  
 for (int i = 0; i < num; ++i) {  
 printf("输出元素结果是:%d\n", arr[i]); // arr[i] 隐士 等价与 \* (arr + i)  
 }  
  
 getchar();  
  
 return 0;  
}

### 5.动态开辟之realloc。

#include <stdio.h>

#include <stdlib.h>  
  
// 动态开辟之realloc  
int main() {  
  
 int num;  
 printf("请输入个数");  
 // 获取用户输入的值  
 scanf("%d", &num);  
  
 // 5个值  
 int \* arr = (int \*) malloc(sizeof(int) \* num);  
 for (int i = 0; i < num; ++i) {  
 arr[i] = (i + 10001); // arr[i]的内部隐士 == \*(arr+i)  
 }  
 printf("开辟的内存指针: %p\n", arr);  
  
 // 打印 内容  
 for (int i = 0; i < num; ++i) {  
 // Derry装B的打印  
 // &取出内存地址 \*然后去值  
 // &取出内存地址 \*然后去值  
 // &取出内存地址 \*然后去值  
 // .....  
 printf("元素的值:%d, 元素的地址:%p\n",  
 \*&\*&\*&\*&\*&\*&\*&\*&  
 \*&\*&\*&\*&\*&\*&\*&\*&\*&  
 \*&\*&\*&\*&\*&\*&\*&\*&  
 \*&\*&\*&\*&\*&\*&\*&\*&  
 \*&\*&\*&\*&\*&\*&\*&\*&  
 \*&\*&\*&\*&  
 \*&\*(arr + i)  
 ,  
 (arr + i)  
 );  
 }  
  
  
  
  
 // ================================= 在堆区开辟新的空间 加长空间大小  
  
 // C的岗位  
 // C工程师的面试题： realloc 为什么一定要传入 arr指针，为什么要传总大小  
  
 // 新增  
 int new\_num;  
 printf("请输入新增加的个数");  
 scanf("%d", &new\_num);  
  
 // 原来的大小4 + 新增加的大小4 = 总大小 8  
 // void \*realloc (void \*前面开辟的指针, size\_t总大小);  
 int \* new\_arr = (int \*) realloc(arr, sizeof(int) \* (num + new\_num));  
  
 if (new\_arr) { // new\_arr != NULL 我才进if 【非0即true】  
 int j = num; // 4开始  
 for (; j < (num + new\_num); j++) { // 5 6 7 8  
 arr[j] = (j + 10001);  
 }  
  
 printf("新 开辟的内存指针: %p\n", new\_arr);  
  
 // 后 打印 内容  
 for (int i = 0; i < (num + new\_num); ++i) {  
  
 printf("新 元素的值:%d, 元素的地址:%p\n",  
 \*(arr + i),  
 (arr + i)  
 );  
 }  
 }  
  
 // 我已经释放  
 free(new\_arr);  
 new\_arr = NULL;  
  
 // 1000行代码  
 // 。。。  
  
 // 重复释放/重复free VS会奔溃， CLion会优化(发现不奔溃) [错误的写法]  
 /\*free(new\_arr);  
 new\_arr = NULL;\*/  
  
 // 必须释放【规则】  
 /\*if (arr) {  
 free(arr); // 如果不赋值给NULL，就是悬空指针了  
 arr = NULL;  
 }\*/  
  
 if (new\_arr) {  
 free(new\_arr); // 如果不赋值给NULL，就是悬空指针了  
 new\_arr = NULL;  
 }  
  
 return 0;  
}

## 指针实战运用进阶之字符串操作

### 1.上节课遗留点完成。

#include <stdio.h>

#include <stdlib.h>  
  
// 动态开辟之realloc  
int mainT0() {  
  
 int num;  
 printf("请输入个数");  
 // 获取用户输入的值  
 scanf("%d", &num);  
  
 // 5个值  
 int \* arr = (int \*) malloc(sizeof(int) \* num);  
 for (int i = 0; i < num; ++i) {  
 arr[i] = (i + 10001); // arr[i]的内部隐士 == \*(arr+i)  
 }  
 printf("开辟的内存指针: %p\n", arr);  
  
 // 打印 内容  
 for (int i = 0; i < num; ++i) {  
 // Derry装B的打印  
 // &取出内存地址 \*然后去值  
 // &取出内存地址 \*然后去值  
 // &取出内存地址 \*然后去值  
 // .....  
 printf("元素的值:%d, 元素的地址:%p\n",  
 \*&\*&\*&\*&\*&\*&\*&\*&  
 \*&\*&\*&\*&\*&\*&\*&\*&\*&  
 \*&\*&\*&\*&\*&\*&\*&\*&  
 \*&\*&\*&\*&\*&\*&\*&\*&  
 \*&\*&\*&\*&\*&\*&\*&\*&  
 \*&\*&\*&\*&  
 \*&\*(arr + i)  
 ,  
 (arr + i)  
 );  
 }  
  
  
  
  
 // ================================= 在堆区开辟新的空间 加长空间大小  
  
 // C的岗位  
 // C工程师的面试题： realloc 为什么一定要传入 arr指针，为什么要传总大小  
  
 // 新增  
 int new\_num;  
 printf("请输入新增加的个数");  
 scanf("%d", &new\_num);  
  
 // 原来的大小4 + 新增加的大小4 = 总大小 8  
 // void \*realloc (void \*前面开辟的指针, size\_t总大小);  
 int \* new\_arr = (int \*) realloc(arr, sizeof(int) \* (num + new\_num));  
  
 if (new\_arr) { // new\_arr != NULL 我才进if 【非0即true】  
 int j = num; // 4开始  
 for (; j < (num + new\_num); j++) { // 5 6 7 8  
 arr[j] = (j + 10001);  
 }  
  
 printf("新 开辟的内存指针: %p\n", new\_arr);  
  
 // 后 打印 内容  
 for (int i = 0; i < (num + new\_num); ++i) {  
  
 printf("新 元素的值:%d, 元素的地址:%p\n",  
 \*(arr + i),  
 (arr + i)  
 );  
 }  
 }  
  
 // 我已经释放  
 /\*free(new\_arr);  
 new\_arr = NULL;\*/  
  
 // 1000行代码  
 // 。。。  
  
 // 重复释放/重复free VS会奔溃， CLion会优化(发现不奔溃) [错误的写法]  
 /\*free(new\_arr);  
 new\_arr = NULL;\*/  
  
 // 必须释放【规则】  
 /\*if (arr) {  
 free(arr); // 如果不赋值给NULL，就是悬空指针了  
 arr = NULL;  
 }\*/  
  
 /\*if (new\_arr) {  
 free(new\_arr); // 如果不赋值给NULL，就是悬空指针了  
 new\_arr = NULL;  
 }\*/  
  
  
 // *TODO ========================== 下面代码对我们上节课 遗留点 完成 末尾工作了* if (new\_arr) { // new\_arr != NULL 进去if， 重新开辟的堆空间是成功的  
 free(new\_arr);  
 new\_arr = NULL;  
 arr = NULL; // 他还在指向那块空间，为了不出现悬空指针，指向NULL的空间  
 } else { // 重新开辟的堆空间是失败的  
 free(arr);  
 arr = NULL;  
 }  
 // 画图：内存的走向 1  
  
  
 return 0;  
}

#include <stdio.h>

// 后续给大家画图(下次画图)  
int mainT1() {  
  
 // 数组  
 int arrInt[] = {6,4,8,3,1,2,9,7,0,5};  
  
 // \*(arrInt+4)) 数组的0元素 + 4 1的地址 \*的值==1 *TODO OK* // arrInt + 前面 ： 数组的0元素 + 1  
  
 int result = arrInt[\*(arrInt+\*(arrInt+4))];  
  
 printf("result的值是多少呀:%d\n", result);  
  
 return 0;  
  
}

### 2.字符串两种形式。

#include <stdio.h>

// 字符串  
int mainT2() {  
 char str[] = {'D', 'e', 'r', 'r', 'y', '\0'};  
 str[2] = 'z'; // 这里能修改？  
 printf("第一种方式：%s\n", str); // printf 必须遇到 \0 才结束  
  
 char \* str2 = "Derry"; // 隐士 Derry+\0  
 str2[2] = 'z'; // 会奔溃，为什么，不允许访问，为什么 画图 原理图  
 printf("第二种方式：%s\n", str);  
  
 return 0;  
}

### 3.指针挪动获取字符串信息(手写API)。

// 3.指针挪动获取字符串信息(手写API)。

#include <stdio.h>  
  
// 获取长度  
/\*  
int getLen(char \* string) {  
 int count = 0;  
 while (\*string) { // \*string != '\0' 我就一直循环  
 string ++;  
 count++;  
 }  
 return count;  
}  
  
int mainT3() {  
 char string[] = {'A', 'B', 'C', 'D', '0', '\0'}; // printf方法能够停下来，不要打印系统值  
  
 int r = getLen(string);  
 printf("长度是:%d\n",r);  
  
 return 0;  
}  
\*/  
  
  
  
#include <stdio.h>  
  
// C/C++编译器 数组作为参数传递，会把数组优化成指针（为了高效率）  
void getLen(int \* resultLen, int intarr[]) { // 模仿了 strLen函数  
  
 // 数组作为参数传递 就不能这样写了  
  
 // sizeof(intarr)28 / sizeof(int)4 = 7  
 /\*int len = sizeof(intarr) / sizeof(int);  
 printf("getLen len长度是:%d\n", len);\*/  
  
 // 手动计算长度  
 int count = 0;  
 while (\*intarr != '\0') {  
 intarr++;  
 count++;  
 }  
 \*resultLen = count;  
}  
  
void doEngine(int len) {  
  
}  
  
int mainT3() {  
 // int 4 char 1  
  
 // char string[] = {'A', 'B', 'C', 'D', '0', '\0'}; // printf方法能够停下来，不要打印系统值  
  
 int intarr[] = {1,2,3, 0,5,6,7}; // 0 \0 其他函数 for 其他调用判断的时候，这个有冲突 【int类型数组】  
  
 // int类型使用这种方式【第一种方式】  
 // sizeof(intarr)28 / sizeof(int)4 = 7  
 int len = sizeof(intarr) / sizeof(int);  
 printf("len长度是:%d\n", len);  
  
 int result; // &取出result遍历的地址给函数  
 getLen(&result, intarr);  
 printf("getLen len长度是:%d\n", result);  
  
 // strLen API 的用法太简单 【第二种方式】 尝试  
  
 // 真实函数直接调用，可以这样，传递长度  
 doEngine(len);  
  
 return 0;  
}

### 4.字符串的比较。

// 4.字符串转换，字符串的比较。

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
// 快捷键：按两下 shift键 选择 file encoding  
  
// 真正的C语言，在Linux上学，在VIM上才是专业的  
int main() {  
 // 字符串转换 =======================  
 char \* num = "1"; // 字符串  
 num = "12.68";  
  
 // 【int】  
 int result = atoi(num);  
 if (result) { // 非0即ture 不是0进入if， 0就是转换失败了  
 printf("恭喜你转换成功:%d\n", result);  
 } else {  
 printf("转换失败！\n");  
 }  
  
 // 【double】  
 double resultD = atof(num);  
 printf("恭喜你转换成功:%lf\n", resultD);  
  
  
 // 字符串的比较 ======================  
 char \* str1 = "Derry";  
 char \* str2 = "derry";  
  
 // int resultC = strcmp(str1, str2); // strcmp = 区分大小写  
 int resultC = strcmpi(str1, str2); // strcmpi = 不区分大小写  
  
 if (!resultC) { // 0代表是相等的， 非0代表是不相等的  
 printf("相等");  
 } else {  
 printf("不相等");  
 }  
  
 return 0;  
}

### 5.字符串查找，包含，拼接。

// 字符串查找，包含，拼接。

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
int mainT5() {  
  
 char \* text = "name is Derry";  
 char \* subtext = "D";  
  
 char \* pop = strstr(text, subtext);  
  
 // 怎么去 字符串查找  
 if (pop) { // 非NULL，就进入if，就查找到了  
 printf("查找到了,pop的值是:%s\n", pop);  
 } else {  
 printf("没有查找到,subtext的值是:%s\n", subtext);  
 }  
  
 // 包含了D吗  
 if (pop) {  
 printf("包含了\n");  
 } else {  
 printf("没有包含\n");  
 }  
  
 // printf("pop地址%p， text地址:%p,\n", pop, text);  
  
 // 求取位置？ 数组是一块连续的内存空间，没有断层，所以可以-  
 int index = pop - text; // pop="Derry" - text"name is Derry"  
 printf("%s第一次出现的位置是:%d\n", subtext, index); // 我的D在第8个位置  
  
 // 指针是可以：++ -- += -=  
  
 // 拼接 ========================  
 char destination[25]; // 容器 25的大小 已经写死了  
 char \* blank = "--到--", \*CPP="C++", \*Java= "Java";  
  
 strcpy(destination, CPP); // 先Copy到数组里面去  
 strcat(destination, blank); // 然后再拼接  
 strcat(destination, Java); // 然后再拼接  
 printf("拼接后的结果:%s\n", destination); // C++--到--Java  
  
 return 0;  
}

### 6.大小写转换(手写API)。

// 大小写转换(手写API)

#include <stdio.h>  
#include <ctype.h>  
  
// 指针的理解  
void lower(char \* dest, char \* name) {  
 char \* temp = name; // 临时指针，你只能操作，临时指针，不能破坏name指针  
 while (\*temp) {  
 \*dest = tolower(\*temp);  
 temp ++; // 挪动指针位置 ++  
 dest ++; // 挪动指针位置 ++ 目的是为了 挪动一个存储一个 挪动一个存储一个 ...  
 }  
 // printf '\0'  
 \*dest = '\0'; // 避免printf打印系统值  
  
 printf("不能破坏 name:%s\n", name); // temp的好处就是，不会破坏name  
}  
  
// 全部变成小写 derry  
int mainT6() {  
 char \* name = "DerrY";  
  
 // 先定义结果  
 char dest[20];  
 lower(dest, name);  
 printf("小写转换后的结构是:%s\n", dest);  
  
 // 作业：  
 /\*char \* str = "Derry is";  
 char \* result;  
 函数(&result, str, 2, 5);  
 printf("%s\n", result); // 最终输出：rry\*/  
  
 // 这样-不成功  
 /\*char \* a = "ABCDEFG";  
 char \* b = "AB";  
 int r = b - a;  
 printf("r:%d\n", r);\*/  
  
 return 0;  
}

## 结构体与结构体指针数组

### 上节课作业讲解

字符串的截取操作：

#include <stdio.h>

#include <string.h>  
#include <stdlib.h>  
  
// *TODO 第一版 通用*void substrAction1(char \* result, char \* str, int start, int end) {  
 char \* temp = str; // 定义临时指针，不破坏str指针  
 int count = 0; // 记录当前的位置  
 while (\*temp) {  
 // 2 到 5 的位置 在截取的范围内  
 if (count >= start && count < end) {  
 \*result = \*temp; // \*temp 取出 D e r r y ...  
 result++; // （接收值也要挪动）挪动指针来接收 = \*temp给我的值  
 }  
 temp++; // 取值也要挪动  
 count++; // 当前的位置要同步  
 }  
}  
  
// *TODO 第二版 有意让同学，深刻理解 栈区 堆区 开辟（1）*void substrAction2(char \*\* result, char \* str, int start, int end) {  
 char \* temp = str; // 定义临时指针，不破坏str指针  
  
 // 合理分配，截取多少用多少，节约  
 char resultArr[end - start]; // 我只需要你截取的大小空间：例如：rry  
  
 // 尽量不要使用第二种方式，会被C工程师鄙视的，为什么？ 你开辟的，就应该你回收  
 // char \* resultArr = malloc(end - start); // 堆区开辟，第二种解决方案  
  
 int count = 0;  
 for (int i = start; i < end; ++i) {  
 resultArr[count] = \*(temp + i); // \*(temp + i);取出D e r r y 给 数组容器  
 count++;  
 }  
  
 // \* 取出二级指针的一级指针 == main函数的result一级指针  
  
 // \*result = resultArr; // 不能让我们的一级指针 指向容器，因为容器会被回收掉  
  
 strcpy(\*result, resultArr); // 第一种解决方案  
  
 printf("%s\n", resultArr);  
  
 // 1 2 不能回收堆空间，否则main函数 打印了空  
 // free(resultArr);  
  
} // 函数弹栈后，会回收所有的栈成员，包括：resultArr  
  
// *TODO 第三版 三行代码搞定*void substrAction3(char \* result, char \* str, int start, int end) { // 没有涉及栈区 堆区的概念  
 // 合理分配，截取多少用多少，节约 思路  
 for (int i = start; i < end; ++i) { // 刚好结束 循环三次  
 \*(result++) = \*(str+i); // i = 2  
 }  
}  
  
// *TODO 第四版 一行代码搞定*void substrAction4(char \* result, char \* str, int start, int end) {  
 // 参数1：我最终是copy到result容器里面  
 // 参数2：直接从r开始，因为我一级做了，指针挪动了  
 // 参数3：你从r开始，挪动多少  
 strncpy(result, str+start, end - start);  
}  
  
// 【截取】字符串的截取操作  
int main() {  
  
 char \*str = "Derry is";  
 // 正好她是一级指针  
 char \*result; // char \* 不需要结尾符\0  
  
 // 截取第二个位置到第五个位置 2，5  
  
 // substrAction1(result, str, 2, 5);  
 // substrAction2(&result,str, 2, 5);  
 // substrAction3(result,str, 2, 5);  
 substrAction4(result, str, 2, 5);  
  
 printf("main 截取的内容是：%s", result); // 最终截取：rry  
  
 if (result) {  
 free(result);  
 result = NULL;  
 }  
  
 return 0;  
}

### 上节课指针练习

#include <stdio.h>

int main() {  
 int a[] = {6,4,8,3,1,2,9,7,0,5};  
  
 // \*(a+4) --- (a+4):a数组默认指向首元素地址 挪动到 第四个元素 \*取出第四个元素内存地址的值 == 1  
 // \*a+1 a数组默认指向首元素地址 挪动到 第一个元素 \*取出第一个元素内存地址的值 == 4  
 // a[4] 挪动到第四个元素 内存地址的值 等于 1  
 int result = a[\*(a+\*(a+4))];  
 printf("result的值是:%d\n", result); // 输出1  
 return 0;  
}

### 结构体定义与使用。

第一种写法：

#include <stdio.h>

#include <string.h>  
#include <stdlib.h>  
#include <malloc.h>  
  
struct Dog {  
 // 成员  
 char name[10]; // copy进去  
 int age;  
 char sex;  
  
}; // 必须给写;  
  
int main() {  
 struct Dog dog; // 这样写完，成员是没有任何初始化的，成员默认值 是系统值(name:?@, age:3133440, sex:€)  
 printf("name:%s, age:%d, sex:%c \n", dog.name, dog.age, dog.sex);  
  
 // 赋值操作  
 // dog.name = "旺财";  
 strcpy(dog.name, "旺财");  
 dog.age = 3;  
 dog.sex = 'G';  
 printf("name:%s, age:%d, sex:%c \n", dog.name, dog.age, dog.sex);  
 return 0;  
}

第二种写法：

#include <stdio.h>

#include <string.h>  
#include <stdlib.h>  
#include <malloc.h>  
  
struct Person {  
 // 成员  
 char \* name; // 字符指针 = "赋值"  
 int age;  
 char sex;  
} ppp = {"Derry", 33, 'M'},  
ppp2,  
ppp3,  
pppp4,  
pppp5  
// ...  
;  
  
int main() {  
  
 // Person == ppp == struct Person ppp;  
 printf("name:%s, age:%d, sex:%c \n", ppp.name, ppp.age, ppp.sex);  
  
 // 赋值  
 // strcpy(pppp5.name, "Derry5"); // Copy不进去  
 pppp5.name = "DerryO";  
 pppp5.age = 4;  
 pppp5.sex = 'M';  
  
 printf("name:%s, age:%d, sex:%c \n", pppp5.name, pppp5.age, pppp5.sex);  
  
  
 return 0;  
}

第三种写法：

#include <stdio.h>

#include <string.h>  
#include <stdlib.h>  
#include <malloc.h>  
  
struct Study {  
 char \* studyContent; // 学习的内容  
};  
  
struct Student {  
 char name[10];  
 int age;  
 char sex;  
  
 // Study study; // VS的写法  
 struct Study study; // Clion工具的写法  
  
 struct Wan {  
 char \* wanContent; // 玩的内容  
 } wan;  
};  
  
int main() {  
  
 struct Student student =  
 {"李元霸", 88, 'm' ,  
 {"学习C"},  
 {"王者农药"}  
 };  
  
 printf("name:%s, age:%d, sex:%c，study:%s, wan:%s \n",  
 student.name, student.age, student.sex, student.study.studyContent, student.wan.wanContent);  
  
  
 return 0;  
}

### 结构体指针与动态内存开辟。

#include <stdio.h>

#include <string.h>  
  
struct Cat {  
 char name[10];  
 int age;  
};  
  
int main() { // 栈  
  
 // 结构体  
 struct Cat cat = {"小花猫", 2};  
  
 // 结构体 指针 -> 调用一级指针成员  
 // VS的写法：Cat \* catp = &cat;  
 struct Cat \* catp = &cat;  
 catp->age = 3;  
 strcpy(catp->name, "小花猫2");  
 printf("name:%s, age:%d \n", catp->name, catp->age);  
  
 return 0;  
}

#include <stdio.h>

#include <string.h>  
#include <stdlib.h>  
  
struct Cat2 {  
 char name[10];  
 int age;  
};  
  
int main() { // 堆  
  
 // VS的写法：Cat2 \* cat = (Cat2 \*) malloc(sizeof(Cat2));  
 struct Cat2 \*cat = malloc(sizeof(struct Cat2));  
  
 strcpy(cat->name, "金色猫");  
 cat->age = 5;  
  
 printf("name:%s, age:%d \n", cat->name, cat->age);  
  
 // 堆区的必须释放  
 free(cat);  
 cat = NULL;  
  
 return 0;  
}

### 结构体的数组。

#include <stdio.h>

#include <stdlib.h>  
#include <string.h>  
  
struct Cat3 {  
 char name[10];  
 int age;  
};  
  
int main() {  
  
 // 栈区 静态范畴  
 struct Cat3 cat [10] = {  
 {"小黄", 1},  
 {"小白", 2},  
 {"小黑", 3},  
 {},  
 {},  
 {},  
 {},  
 {},  
 {},  
 {},  
 };  
  
 // VS的写法  
 // cat[9] = {"小黑9", 9},  
  
 // ClION的写法  
 struct Cat3 cat9 = {"小黑9", 9};  
 // cat[9] = cat9;  
 \*(cat + 9) = cat9;  
 printf("name:%s, age:%d \n", cat9.name, cat9.age);  
   
 // 堆区 动态范畴 ==============================  
 struct Cat3 \* cat2 = malloc(sizeof(struct Cat3) \* 10);  
  
 // 【1元素地址的操作】给他赋值，请问是赋值，那个元素 （默认指向首元素地址）  
 strcpy(cat2->name, "小花猫000");  
 cat2->age = 9;  
 printf("name:%s, age:%d \n", cat2->name, cat2->age);  
  
 // 【8元素地址的操作】 给第八个元素赋值  
 cat2 += 7;  
 strcpy(cat2->name, "小花猫888");  
 cat2->age = 88;  
 printf("name:%s, age:%d \n", cat2->name, cat2->age);  
  
 free(cat2);  
 cat2 = NULL;  
  
  
 return 0;  
}

### 结构体与结构体指针取别名。

#include <stdio.h>

#include <stdlib.h>  
  
struct Workder\_ {  
 char name[10];  
 int age;  
 char sex;  
};  
  
// VS的写法：typedef Workder\_  
typedef struct Workder\_ Workder\_; // 给结构体取别名  
  
typedef Workder\_ \* Workder; // 给结构体指针取别名  
  
// C库的源码，系统源码...，为什么 typedef 还取一个和结构体一样的名字（兼容代码的写法，保持一致）  
  
int main() {  
 // 以前 Clion工具 必须加上 struct VS又不用加 代码差异化大  
 // struct Workder\_ workder1 = malloc(sizeof(struct Workder\_));  
  
 // 现在 （兼容代码的写法，保持一致）  
 Workder\_ workder1 = malloc(sizeof(Workder\_));  
  
 // VS CLion 他们都是一样的写法  
 Workder workder = malloc(sizeof(Workder\_));  
  
 return 0;  
}

// 系统源码都是这样写的

#include <stdio.h>  
#include <stdlib.h>  
  
struct DAO {  
 char name[10];  
 int age;  
 char sex;  
};  
  
// 匿名结构体的别名（这样写意义不大，因为没有名字）  
typedef struct {  
 char name[10];  
 int age;  
 char sex;  
};  
  
// 源码是这样写的  
// 给结构体AV 取了一个别名等于AV  
typedef struct {  
 char name[10];  
 int age;  
 char sex;  
} AV;  
  
// 取一个别名  
typedef struct DAO DAO;  
  
void show(DAO dao) {} // 在不同工具上 又的要加，又的不用加 又差异化  
  
int main() {  
 // VS 不需要这样写， Clion工具 要加入关键字 代码不统一  
 // struct DAO \* dao = malloc( sizeof(struct DAO));  
  
 // 加别名后 代码的统一了  
  
 // VS  
 DAO \* dao = malloc( sizeof(DAO));  
  
 // CLion工具也这样写  
 DAO \* dao1 = malloc( sizeof(DAO));  
  
 // xxx 工具也这样写  
 DAO \* dao2 = malloc( sizeof(DAO));  
  
 // 加别名后 代码的统一了  
 // C库的源码，系统源码...，为什么 typedef 还取一个和结构体一样的名字（兼容代码的写法，保持一致）  
 AV av = {"VideoInfo", 54, 'M'}; // 结构体 VS Clion xxx工具 兼容写法  
  
 AV \* avp = malloc(sizeof(AV)); // 结构体指针  
  
 return 0;  
}

### 枚举。

#include <stdio.h>

// 枚举 int 类型的  
enum CommentType {  
 *TEXT* = 10,  
 *TEXT\_IMAGE*,  
 *IMAGE*};  
  
// 作业：处理好 差异化代码  
int main() {  
 // Clion工具的写法如下：  
 enum CommentType commentType = *TEXT*;  
 enum CommentType commentType1 = *TEXT\_IMAGE*;  
 enum CommentType commentType2 = *IMAGE*;  
  
 // VS工具的写法如下：  
 // CommentType commentType = TEXT;  
  
 printf("%d, %d, %d \n", commentType, commentType1, commentType2);  
  
 return 0;  
}

## C语言游戏破解器与文件加解密

### 1.上节课作业

// 上节课作业

#include <stdio.h>  
  
// 枚举在创建的时候已经取别名了  
typedef enum {  
 *AUDIO* = 111,  
 *VIDEO*,  
 *INFO*} AV;  
  
int main() {  
 printf("截拳道");  
  
 // 以后写C代码的习惯，尽量加上别名，来保证代码的 统一  
 AV av1 = *AUDIO*;  
 AV av2 = *VIDEO*;  
 AV av3 = *INFO*;  
 printf("audio:%d, video:%d, info:%d,\n", av1, av2, av3);  
 // 截拳道audio:111, video:112, info:113,  
  
 return 0;  
}

游戏作弊器

// 游戏作弊器

// C语言：地址的重要性  
  
#include <stdio.h>  
#include <windows.h>  
  
int main() {  
 printf("游戏倒计时开始了....\n");  
  
 // Linux就不规范  
 int i;  
 for (i = 100; i > 0; i--) {  
 printf("还剩下多少秒:%d, 内存地址是:%p\n", i, &i);  
 Sleep(20000);  
 }  
  
 printf("恭喜，你已经赢得了游戏最高等级，碉堡了!\n");  
  
 getchar(); // 停留  
  
 // 最难的是地址的寻找，只要你能找到地址，就操作一切  
  
 return 0;  
}

### 2.文件的读。

// 2.文件的读。

#include <stdio.h>  
#include <stdlib.h> // 文件的操作，是在这个头文件里面的  
#include <string.h>  
  
int main() {  
  
 // fopen打开文件的意思（参数1：文件路径 文件源， 参数2：模式 r(读) w(写) rb(作为二进制文件读) rw(作为二进制文件写) 返回值 FILE 结构体）  
 // FILE \* fopen (const char \*, const char \*);  
  
 char \* fileNameStr = "D:\\Temp\\DerryFile.txt";  
  
 // 既然是使用了r，你要提前准备好文件  
 FILE \* file = fopen(fileNameStr, "r");  
  
 if (!file) {  
 printf("文件打开失败，请你个货检测：路径为%s路径的文件，看看有什么问题\n", fileNameStr);  
 exit(0); // 退出程序  
 }  
  
 // 先定义缓存区域 (容器)  
 char buffer[10];  
  
 // 1.缓冲区buffer， 2：长度10， 3:文件指针变量  
 while (fgets(buffer, 10, file)) {  
 printf("%s", buffer);  
 }  
  
 // 关闭文件  
 fclose(file);  
  
 return 0;  
}

### 3.文件的写。

// 3.文件的写。

#include <stdio.h>  
#include <stdlib.h> // 文件的操作，是在这个头文件里面的  
#include <string.h>  
  
int main() {  
  
 // fopen打开文件的意思（参数1：文件路径 文件源， 参数2：模式 r(读) w(写) rb(作为二进制文件读) rw(作为二进制文件写) 返回值 FILE 结构体）  
 // FILE \* fopen (const char \*, const char \*);  
  
 char \* fileNameStr = "D:\\Temp\\DerryFileW.txt";  
  
 // 既然是使用了w，他会自动生成文件 0kb  
 FILE \* file = fopen(fileNameStr, "w");  
  
 if (!file) {  
 printf("文件打开失败，请你个货检测：路径为%s路径的文件，看看有什么问题\n", fileNameStr);  
 exit(0); // 退出程序  
 }  
  
 fputs("Derry Success run...", file);  
  
 // 关闭文件  
 fclose(file);  
  
 return 0;  
}

### 4.文件复制。

// 4.文件的复制。

#include <stdio.h>  
#include <stdlib.h> // 文件的操作，是在这个头文件里面的  
#include <string.h>  
  
// 二进制文件来复制 rb rw  
int main() {  
  
 // https://en.cppreference.com/w/c/io  
  
 char \* fileNameStr = "D:\\Temp\\DerryFile.txt"; // 来源  
 char \* fileNameStrCopy = "D:\\Temp\\DerryFileCopy.txt"; // 目标  
  
 // rb 读取二进制数据  
 FILE \* file = fopen(fileNameStr, "rb");  
  
 // rw 写入二进制数据  
 FILE \* fileCopy = fopen(fileNameStrCopy, "wb");  
  
 if (!file || !fileCopy) {  
 printf("文件打开失败，请你个货检测：路径为%s路径的文件，看看有什么问题\n", fileNameStr);  
 exit(0); // 退出程序  
 }  
  
 int buffer[514]; // 514 \* 4 = 2048  
 int len; // 每次读取的长度  
  
 // fread：参数1：容器buffer， 参数2：每次偏移多少 int， 参数3：容器大小 写个2048，等下文件就报废了  
 // sizeof(buffer) / sizeof(int) 等价与 514  
 while ((len = fread(buffer, sizeof(int), 514 , file)) != 0) {  
 fwrite(buffer, sizeof(int), len, fileCopy);  
 }  
  
 // 关闭文件  
 fclose(file);  
 fclose(fileCopy);  
  
 return 0;  
}

### 5.获取文件的大小。

// 文件大小的获取

#include <stdio.h>  
#include <stdlib.h> // 文件的操作，是在这个头文件里面的  
#include <string.h>  
  
int main() {  
  
 // 没有专门的 文件大小获取 API，  
 // （思路：）读取头指针 头指针挪动位置，挪动到最后，就可以求得文件大小  
  
 char \* fileNameStr = "D:\\Temp\\DerryFile.txt"; // 来源  
  
 // 既然是使用了w，他会自动生成文件 0kb  
 FILE \* file = fopen(fileNameStr, "rb"); // file == 指针  
  
 if (!file) {  
 printf("文件打开失败，请你个货检测：路径为%s路径的文件，看看有什么问题\n", fileNameStr);  
 exit(0); // 退出程序  
 }  
  
 // 都是操作指针来玩的  
  
 // SEEK\_SET（开头） SEEK\_CUR（当前） SEEK\_END（结尾）  
 fseek(file, 0, SEEK\_END);  
 // 走到这里之后：file有了更丰富的值，给你的file指针赋值，挪动的记录信息  
  
 // 读取 刚刚给file赋值的记录信息  
 // 其实此函数目的是：计算偏移的位置,ftell 从 0 开始统计到当前（SEEK\_END）  
 long file\_size = ftell(file);  
 printf("%s文件的字节大小是:%ld\n", fileNameStr, file\_size);  
 // 8 字节 (8 字节)  
  
 // 关闭文件  
 fclose(file);  
  
 return 0;  
}

### 6.文件加密与文件解密。

// *TODO 【文件的加密】*

// 文件加密 与 文件解密 二进制的方式来操作 .txt .png ...  
  
#include <stdio.h>  
#include <stdlib.h> // 文件的操作，是在这个头文件里面的  
#include <string.h>  
  
int main() {  
 char \* fileNameStr = "D:\\Temp\\IMG.jpg"; // 来源  
 char \* fileNameStrEncode = "D:\\Temp\\IMG\_encode.jpg"; // 加密后的目标文件  
  
 FILE \* file = fopen(fileNameStr, "rb"); // r 必须字节提前准备好文件  
 FILE \* fileEncode = fopen(fileNameStrEncode, "wb"); // w 创建一个0kb的文件  
  
 if (!file || !fileEncode) {  
 printf("文件打开失败，请你个货检测：路径为%s路径的文件，看看有什么问题\n", fileNameStr);  
 exit(0); // 退出程序  
 }  
  
 // 【加密 和 解密 的 思路】  
 // 加密 == 破坏文件  
 // 解密 == 还原文件 还原之前的样子  
  
 // ---  
 // 加密的想法：（全部）把每一个字节都拿出来，对每一个字节都全部处理， （部分加密：把某一部分内容拿出来处理）  
 // *TODO 加密的想法* // while 循环遍历的时候 取出来的 10  
 // 10 ^异惑 5 就变成这个效果1111  
  
 /// *TODO 解密的想法* // while 循环遍历的时候 取出来的 1111  
 // 1111 ^异惑 5 就变成这个效果10（还原）  
  
 int c; // 接收读取的值  
  
 // fgetc(文件指针) 返回值 EOF = end fo file  
 while ((c = fgetc(file)) != EOF) {  
 // 加密操作  
 fputc(c ^ 5, fileEncode ); // 写入到 fileEncode D:\Temp\IMG\_encode.jpg（加密后的图片）  
 }  
  
 // 关闭文件  
 fclose(file);  
 fclose(fileEncode);  
  
 return 0;  
}

// *TODO 【文件的解密】*

#include <stdio.h>  
#include <stdlib.h> // 文件的操作，是在这个头文件里面的  
#include <string.h>  
  
int main() {  
  
 char \*fileNameStr = "D:\\Temp\\IMG\_encode.jpg"; // 来源  
 char \*fileNameStrDecode = "D:\\Temp\\IMG\_decode.jpg"; // 解密后的目标文件  
  
 FILE \* file = fopen(fileNameStr, "rb"); // r 必须字节提前准备好文件  
 FILE \* fileEncode = fopen(fileNameStrDecode, "wb"); // w 创建一个0kb的文件  
  
 if (!file || !fileEncode) {  
 printf("文件打开失败，请你个货检测：路径为%s路径的文件，看看有什么问题\n", fileNameStr);  
 exit(0); // 退出程序  
 }  
  
 int c;  
 while ((c = fgetc(file)) != EOF) {  
 // 解密操作 1111 ^ 5 = 10；（还原）  
 fputc(c ^ 5, fileEncode);  
 }  
  
 fclose(file);  
 fclose(fileEncode);  
  
 return 0;  
}

// *TODO 【文件的加密，使用密钥】*

#include <stdio.h>  
#include <stdlib.h> // 文件的操作，是在这个头文件里面的  
#include <string.h>  
  
int main() {  
 char \*fileNameStr = "D:\\Temp\\IMG.jpg"; // 来源  
 char \* fileNameStrEncode = "D:\\Temp\\IMG\_encode.jpg"; // 加密后的目标文件  
  
 // 密钥  
 char \* password = "123456"; // 我现在做的事情：我的密钥 必须生效  
  
 FILE \* file = fopen(fileNameStr, "rb"); // r 必须字节提前准备好文件  
 FILE \* fileEncode = fopen(fileNameStrEncode, "wb"); // w 创建一个0kb的文件  
  
 if (!file || !fileEncode) {  
 printf("文件打开失败，请你个货检测：路径为%s路径的文件，看看有什么问题\n", fileNameStr);  
 exit(0); // 退出程序  
 }  
  
 int c;  
 int index = 0;  
 int pass\_len = strlen(password);  
 while ((c = fgetc(file)) != EOF) {  
 char item = password[index%pass\_len];  
 // 1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6 ...  
  
 printf("item:%c%\n", item);  
 fputc(c ^ item, fileEncode);  
 index ++;  
 }  
  
 // 关闭文件  
 fclose(file);  
 fclose(fileEncode);  
}

// *TODO 【文件的解密，使用密钥】*

#include <stdio.h>  
#include <stdlib.h> // 文件的操作，是在这个头文件里面的  
#include <string.h>  
  
int main() {  
  
 char \* fileNameStr = "D:\\Temp\\IMG\_encode.jpg"; // 来源  
 char \* fileNameStrDecode = "D:\\Temp\\IMG\_decode.jpg"; // 解密后的目标文件  
  
 FILE \* file = fopen(fileNameStr, "rb"); // r 必须字节提前准备好文件  
 FILE \* fileDecode = fopen(fileNameStrDecode, "wb"); // w 创建一个0kb的文件  
  
 if (!file || !fileDecode) {  
 printf("文件打开失败，请你个货检测：路径为%s路径的文件，看看有什么问题\n", fileNameStr);  
 exit(0); // 退出程序  
 }  
  
 // 密钥  
 char \* password = "123456"; // 我现在做的事情：我的密钥 必须生效  
  
 int c;  
 int index = 0;  
 int pass\_len = strlen(password);  
 while ((c = fgetc(file)) != EOF) {  
 fputc(c ^ password[index%pass\_len], fileDecode);  
 index++;  
 }  
  
 fclose(file);  
 fclose(fileDecode);  
  
 return 0;  
}

### 7.字符串密码加密与解密

#include <stdio.h>

void substrAction3(char \* result, char \* str, int start, int end) {  
 for (int i = start; i < end; ++i) {  
 \*(result++) = \*(str + i);  
 }  
}  
  
int main() {  
 char \* str = "Derry is";  
  
 // char \* result = "ABCDEFG"; (会报错，因为不准修改常量值)  
  
 // char \* result = NULL; (会报错，因为不准修改NULL值)  
 // VS不允许野指针（严格）  
  
 // char \* result; // 这就是不通过的 才对的 VS 不通过对的  
  
 // 在你的栈区开辟空间[推荐方式]  
 char result[100] = "ABC";  
  
 substrAction3(result, str, 3, 5);  
 printf("%s\n", result);  
  
  
 // 这两个地方搞晕了，  
 // char arry1[]={'1','2'}; // 不会自动增加\0  
 // 一级指针，字符指针，数组  
 char \* arry = "21"; // 会自动增加\0  
 // char arry[10] = "12";  
 char arry3[10] = "12";  
  
 return 0;  
}

## C++语言学习之面向对象

### 1.C语言与C++语言有何区别。

// #include <stdio.h> // C语言的标准支持

#include <iostream> // C++标准支持 C++的与众不同  
  
using namespace std; // 命名空间 C++ 的特性 （Java语言的内部类）  
  
int main() {  
 // C++语言面向对象 + 标准特性  
 // C语言面向过程，函数+结构体  
 // C++里面可以运行C语言，可以调用C语言，反之 就不行C语言无法运行C++  
 // 以后我们85%以上 都是 用C++去写功能  
 // 小故事：谣言 C++-- 在C语言上增加好的内容  
  
 // C++里面可以运行C语言，可以调用C语言，反之 就不行C语言无法运行C++  
 printf("降龙十八掌(C版)\n");  
  
 // std::cout << "C++语言的学习" << std::endl;  
 cout << "C++语言的学习" << endl; // 因为你前面引入了命名空间，省略std::  
  
 // endl == \n 都是换行的含义一样  
  
 // Kotlin也有操作符重载， Kotlin就是各个语言的精华所在  
  
 // << 不是属性里面的运算，操作符重载，后面会讲  
 cout << "擒龙功" << endl;  
  
 cout << "铁头功\n"  
 << "金刚腿\n"  
 << "铁布衫\n";  
  
 return 0;  
}

//// #include <stdio.h> // C语言的标准支持

//  
//#include <iostream> // C++标准支持 C++的与众不同  
//  
//using namespace std; // 命名空间 C++ 的特性 （Java语言的内部类）  
//  
//int main() {  
//  
// return 0;  
//}  
//  
//  
  
// C不能运行C++的代码

### 2.常量之C的常量与C++的常量。

#include <stdio.h>

// C语言的常量，其实是个 假常量（伪命题）  
int main() {  
 const int number = 100;  
  
 // number = 200;  
  
 int \* numP = &number;  
  
 \*numP = 10000;  
  
 printf("%d\n", number);  
  
 return 0;  
}

#include <iostream>

// C++语言的常量，其实是真常量  
int main() {  
 const int number = 100;  
  
 // number = 200;  
  
 // 我的编译器，编译不通过， 有的编译器，编译通过，但是运行报错（结论：就不能改）  
 // int \* numP = &number;  
  
 // \*numP = 10000;  
  
 printf("%d\n", number);  
  
 return 0;  
}

### 3.引用的原理与常量引用。

// 3.引用的原理与

#include <iostream>  
using namespace std;  
  
// 互换两个数  
  
// 指针取地址 互换 C语言第一节课的内容  
// 接收number1/number2的地址，取改地址的值，来完成的互换  
void numberChange(int \* number1, int \* number2) {  
 int temp = 0;  
 temp = \*number1;  
 \*number1 = \*number2;  
 \*number2 = temp;  
}  
  
// C++提倡的引用  
void numberChange2(int & number1, int & number2) {  
  
 // 如果不采用引用，main numberChange2 内存地址是不一样的  
 // 如果采用引用，main numberChange2 内存地址是一样的，为什么呢？  
 cout << "numberChange2 " << "n1地址：" << &number1 << " , n2地址：" << &number2 <<endl;  
  
 int temp = 0;  
 temp = number1;  
 number1 = number2;  
 number2 = temp;  
}  
  
int main() {  
  
 int number1 = 10;  
 int number2 = 20;  
  
 cout << "main " << "n1地址：" << &number1 << " , n2地址：" << &number2 <<endl;  
  
 // numberChange(&number1, &number2);  
 numberChange2(number1, number2);  
  
 cout << "n1:" << number1 << " , n2:" << number2 << endl;  
  
 cout << endl;  
  
 // 引用做实验，来学原理：  
  
 // 第一部分，不采用 &  
 /\*int n1 = 999;  
 int n2 = n1;  
 cout << &n1 << "---" << &n2 << endl;\*/ // 0xffffcbb4---0xffffcbb0  
  
 // 第二部分，采用&  
 int n1 = 999;  
 int & n2 = n1;  
 int & n9 = n1;  
 n2 = 777;  
 n9 = 9527;  
 cout << "地址：" << &n1 << "---" << &n2 << endl;  
 cout << "值：" << n1 << "---" << n2 << endl;  
  
 return 0;  
}

// 常量引用。

#include <iostream>  
#include <string.h>  
  
using namespace std;  
  
// 代码的统一性  
typedef struct {  
 char name[20];  
 int age;  
}Student;  
  
// 常量引用：Student不准你改 == const Student &  
// 插入数据库，Student的信息给插入数据库  
void insertStudent(const Student & student) {  
 // 内鬼 卧底  
 // strcpy(student.name, "李元霸"); 不能这样修改  
  
 Student student2 = {"刘奋", 43};  
 // student = student2; 不能这样修改  
  
 // 只读的了，可以安心插入数据库了  
 cout << student.name << "," << student.age << endl;  
}  
  
int main() {  
  
 // 用户提交的Student数据  
 Student student = {"张无忌", 30};  
 insertStudent(student);  
  
 return 0;  
}

### 4.函数重载与默认行参，无形参名的特殊写法。

// C语言的函数重载 是不支持的

#include <stdio.h>  
  
void add(int number1, int number2) {  
  
}  
  
// error: conflicting types for 'add' == 已有函数主体  
// C语言里面，搞了函数重载，这是不允许的，C++才支持函数重载  
/\*void add(int number1, int number2, int number3) {  
  
}\*/  
  
int main() {  
  
 return 0;  
}

// C++语言的函数重载 是支持的

#include <iostream>  
  
using namespace std;  
  
// Java构造函数 50个字段，我真正使用的，只有6个  
// 建造者设计模式 去解决此问题  
  
// int add(int number1, int number2) = number1 + number2; // 这个是KT写法  
  
int add(int number1) {  
 return number1;  
}  
  
int add(int number1, int number2) {  
 return number1 + number2;  
}  
  
// C++重载 == Java重载  
int add(int number1, int number2, int number3) {  
 return number1 + number2 + number3;  
}  
  
// 函数重载 二义性  
// 自己做实验  
// 默认行参赋值， // KT也有默认行参赋值 优先寻找默认行参赋值的函数， 跟顺序无关  
int add(double n1 = 100, int n2 = 200, int n3 = 300, int n4 = 400, bool isOK = 0) {  
 return 0;  
}  
  
// 后面：函数顺序的问题？  
  
int main() {  
 add(999);  
  
 add(999, 777);  
  
 add(100, 200, 888);  
  
 return 0;  
}

// 系统源码里面大量的写法 [特殊写法，意义何在]

#include <iostream>  
  
using namespace std;  
  
// 前期先抽象出现，我后面再升级  
// 后面你可以扩展  
void JNIMethod(double, double, int, int) {  
  
}  
  
// 上层日志  
// 我前期的时候，没有考虑好，为了防止扩展功能，你必须传递一个int类型参数  
// 0：服务器同步， 1：友萌服务器同步 2：服务器同步+友萌服务器同步  
void uploadLogToEngine(char \* logText, int mode) {  
 // 普通上传日志  
  
 // 半年过后，我再来补这个功能  
 if (mode) {  
  
 } else if (mode == 1) {  
  
 }  
}  
  
int main(void) {  
 // 前面一个月开发功能的时候  
 uploadLogToEngine("xxxxx", 0); // 300出地方调用  
 uploadLogToEngine("xxxxx", 2); // 600出地方调用  
 uploadLogToEngine("xxxxx", 1); // 400出地方调用  
  
 JNIMethod(9.0 ,9.9, 1, 2);  
  
 return 0;  
  
}

### 5.C++面向对象。

// 5.C++面向对象。

#include "Student.h"  
  
int main() {  
 // 规范写法：要有 头文件.h .hpp -- 实现文件 .c cpp  
  
 std::cout << 1 << std::endl;  
  
  
 // *TODO ======= 下面是栈空间* Student student1; // 栈区开辟空间的  
  
 // 赋值  
 student1.setAge(99);  
 student1.setName("李连杰");  
  
 cout << "name:" << student1.getName() << " ,age:" << student1.getAge() << endl;  
  
  
 // *TODO ======= 下面是堆空间* Student \* student2 = new Student(); // new/delete  
  
 // 赋值  
 student2->setAge(88);  
 student2->setName("李元霸");  
  
 cout << "name:" << student2->getName() << " ,age:" << student2->getAge() << endl;  
  
 if (student2)  
 delete student2; // 必须手动释放堆空间的对象student2  
 student2 = NULL; // 指向NULL的地址区域  
 // free(student2); // 不能这样写，不规范，会被鄙视的  
  
 return 0;  
  
  
 // 以后我们学习C++的时候，Derry可能没有写头文件，是不规范的，我只是为了讲课  
 // 真实开发过程中，必须规范来：  
 // 正规的流程：【xxx.so（C/Cpp的实现代码） 用户拿到xxx.so】， 头文件  
  
  
} // main函数弹栈后，会释放栈成员 student1

#include <iostream>

using namespace std; // std 是系统的，必须这样叫， 下节课讲 自定义命名空间  
  
int main() {  
  
 // bool isOK = 1; // !0 就true  
 // bool isOK = -8891; // !0 就true  
 // bool isOK = 4359; // !0 就true  
 bool isOK = 0; // 0 就false  
  
 // 只有两个类型： !0 0  
 // 和前面的C一模一样  
  
 if (isOK) {  
 cout << "真" << endl;  
 } else {  
 cout << "假" << endl;  
 }  
  
 // ==========================  
 bool isRun = true;  
 isRun = false;  
 cout << isRun << endl; // true==1, false=0  
  
 return 0;  
}

## C++语言类中各个重要函数原理

### 命名空间：

// 命名空间

#include <iostream>  
  
// 声明std，我们的main函数就可以直接使用里面的成员，不需要使用 std::  
using namespace std; // C++自己的命名空间 (C# .net 命名空间)  
  
// 自定义命名空间  
namespace derry1 {  
 int age = 33;  
 char \* name = "Derry猛男1";  
  
 void show() {  
 cout << "name:" << name << ", age:" << age << endl;  
 }  
  
 void action() {  
 cout << "derry1 action" << endl;  
 }  
}  
  
// *TODO ------ 命名空间里面重复的函数*// 自定义命名空间  
namespace derry2 {  
 void action() {  
 cout << "derry2 action" << endl;  
 }  
}  
  
// *TODO ------ 小概率会遇到的情况，命名空间的嵌套*// 自定义命名空间  
namespace derry3 {  
 namespace derry3Inner {  
 namespace derry3Inner1 {  
 namespace derry3Inner2 {  
 namespace derry3Inner3 {  
 void out() {  
 cout << "爱恨情仇人消瘦，悲欢起落人寂寞" << endl;  
 }  
 }  
 }  
 }  
 }  
}  
  
// 声明各个写的 命名空间  
// using namespace derry1;  
  
int main() {  
 cout << "命名空间" << endl;  
  
 // 声明各个写的 命名空间  
 using namespace derry1;  
  
 int ageValue = derry1::age; // 方式1 使用 刚刚声明的命名空间  
 derry1::show(); // 使用 刚刚声明的命名空间  
  
 ageValue = age; // 方式2 直接去引出来 ::  
 show(); // 直接去引出来 ::  
  
  
 // *TODO ------ 命名空间里面重复的函数* using namespace derry2;  
 // action(); 很尴尬  
 derry1::action();  
 derry2::action();  
  
 // *TODO ------ 小概率会遇到的情况，命名空间的嵌套* // 第一种方式 先声明命名空间 再使用  
 using namespace derry3::derry3Inner::derry3Inner1::derry3Inner2::derry3Inner3;  
 // 再使用  
 out();  
  
 // 第二种方式 直接使用  
 derry3::derry3Inner::derry3Inner1::derry3Inner2::derry3Inner3::out();  
  
 return 0;  
}

### 构造函数，析构函数详讲。

// 1.构造函数详讲， 2.析构函数， 3.Java/KT的所谓的析构函数。

#include <iostream>  
#include <string.h>  
using namespace std;  
  
class Student {  
  
// 构造函数  
public:  
 // 空参数构造函数  
 Student() {  
 cout << "空参数构造函数" << endl;  
 }  
  
 // 一个参数的构造函数  
 // :Student(name, 87) 等价 1.调用两个参数的构造函数， 2.再调用当前函数  
 // 学生说的：先调用两个参数的，再调用一个的  
 Student(char \*name) :Student(name, 87) {  
 cout << "一个参数的构造函数" << endl;  
 this->name = name;  
 }  
 // 系统源码中是写的  
 // :name(name) 等价 this->name = name;  
 /\*Student(char \* name) :name(name) {  
 cout << "一个参数的构造函数" << endl;  
 }\*/  
  
 // 两个参数的构造函数  
 Student(char \*name, int age) {  
 // this->name = name;  
  
 // 堆区  
 this->name = (char \*) (malloc(sizeof(char \*) \* 10));  
 strcpy(this->name, name);  
  
 this->age = age;  
 cout << "两个参数的构造函数" << endl;  
 }  
  
 // 析构函数 Student对象的，临终遗言，Student对象被回收了，你做一些释放工作  
 // delete stu 的时候，我们的析构函数一定执行  
 // free不会执行析构函数，也意味着，你没法在析构函数里面，做释放工作， malloc也不会调用构造函数  
 ~Student() {  
 cout << "析构函数" << endl;  
  
 // 必须释放 堆区开辟的成员  
 if (this->name) {  
 free(this->name);  
 this->name = NULL; // 执行NULL的地址，避免出现悬空指针  
 }  
 }  
  
// 私有属性  
private:  
 char \*name;  
 int age;  
  
// 公开的 set get 函数  
public:  
 int getAge() {  
 return this->age;  
 }  
  
 char \*getName() {  
 return this->name;  
 }  
  
 void setAge(int age) {  
 this->age = age;  
 }  
  
 void setName(char \*name) {  
 this->name = name;  
 }  
};  
  
int main() {  
 // *TODO =========== 下面是栈区 开辟空间的* /\*Student stu; // 调用 空参数构造函数  
 stu.setAge(34);  
 stu.setName("李元霸");  
 cout << "name:" << stu.getName() << ", age:" << stu.getAge() << endl;\*/  
  
 // Student stu("雄霸", 30);  
 /\*Student stu("李连杰");  
 cout << "name:" << stu.getName() << ", age:" << stu.getAge() << endl;\*/  
  
 // *TODO =========== 下面是堆区 开辟空间的 堆区必须手动释放，否则内存占用越来* // 系统源码中，会看到，很多使用 new 关键字  
  
 // \*stu ->：调用一级指针的成员  
 // new/delete  
 // C++中，必须使用 new/delete 一套  
 Student \*stu = new Student("杜子腾", 26);  
 cout << "name:" << stu->getName() << ", age:" << stu->getAge() << endl;  
 delete stu;  
  
 // free(stu); 这样写是不规范的，不按人家规则来  
  
 // 还有人这样写 (C工程师，搞了六年，改不了自己的习惯， malloc)  
 // malloc 你的构造函数都没有调用，这个不行的  
 /\*Student \*stu2 = (Student\*) malloc(sizeof(Student));  
 free(stu2);\*/  
  
 // 纠结：C++有析构函数（临终遗言 释放工作） Java KT 临终遗言 是什么？  
  
 return 0;  
}

### 纠结new/delete。

// 3.纠结new/delete。

// 以后必须很牢固  
  
// new/delete 是一套 会调用构造函数 与 析构函数 【C++标准规范】  
  
// malloc/free是一套 不调用构造函数 与 析构函数 【C的范畴，虽然不推荐，但是也是可以的】

3.拷贝构造函数。

// 4.拷贝构造函数。

#include <iostream>  
#include <string.h>  
  
using namespace std;  
  
class Student {  
  
public:  
 Student() { cout << "空参数构造函数" << endl; }  
  
 // 两个参数的构造函数  
 Student(char \*name, int age) : name(name), age(age) {  
 cout << "两个参数构造函数" << endl;  
 }  
  
 // 析构函数  
 // ~Student(char \* name) { } 这样写，就不是析构函数了，如果你这样写，C/C++编译器对你很无语  
 ~Student() {  
 cout << "析构函数" << endl;  
 }  
  
 // 以前是默认有一个拷贝构造函数，stu2 = stu1 默认赋值 【隐士 你看不到】  
  
 // 拷贝构造函数，它默认有，我们看不到，一旦我们写拷贝构造函数，会覆盖她  
 // 对象1 = 对象2  
 // 覆盖拷贝构造函数  
 Student(const Student & student) { // 常量引用：只读的，不让你修改  
 cout << "拷贝构造函数" << endl;  
  
 // 我们自己赋值  
 // 为什么要自己赋值，自己来控制，就可以 例如：-10  
 this->name = student.name;  
 this->age = student.age - 10;  
  
 cout << "自定义拷贝构造函数 内存地址 " << &student << endl;  
 }  
  
// 私有属性  
private:  
 char \*name;  
 int age;  
  
// 公开的 set get 函数  
public:  
 int getAge() {  
 return this->age;  
 }  
  
 char \*getName() {  
 return this->name;  
 }  
  
 void setAge(int age) {  
 this->age = age;  
 }  
  
 void setName(char \*name) {  
 this->name = name;  
 }  
};  
  
struct Person {  
 int age;  
 char \*name;  
};  
  
// *TODO = 号的意义 隐士代码，引出 拷贝构造函数*/\*  
int main() {  
 Person person1 = {100, "张三丰"};  
  
 // = 你看起来，没有什么特殊，隐士的代码：你看不到 C/C++编译器 会把p1的成员值赋值给p2成员  
 Person person2 = person1;  
  
 cout << &person1 << endl;  
 cout << &person2 << endl;  
  
 cout << person2.name << ", " << person2.age << endl;  
  
 // 思考：对象 对象1=对象2 默认的 拷贝构造函数  
  
 return 0;  
}\*/  
  
// *TODO 拷贝构造函数*/\*  
int main() {  
 Student stu1("李鬼", 34);  
 Student stu2 = stu1;  
  
 cout << stu2.getName() << " , " << stu2.getAge() << endl;  
 cout << "main " << &stu1 << endl; // 地址的打印是一样的， 注意：cnetos的环境 地址打印有差异，要注意  
  
  
  
 // *TODO 拷贝构造函数的注意点：*  
 // Student stu1("李鬼", 34);  
  
 // Student stu2;  
 // stu2 = stu1; // 这样赋值是不会调用 自定义拷贝构造函数，但是会调用默认赋值  
 // Student stu2 = stu1; // 这样赋值是会调用 自定义拷贝构造函数，我们自己赋值  
  
 // cout << stu2.getName() << " , " << stu2.getAge() << endl;  
  
 getchar(); // 程序等待在这一行  
  
 return 0;  
} // main函数弹，stu1栈成员会回收 stu2栈成员会回收  
\*/  
  
// *TODO 这种写法 拷贝构造函数 到底会不会调用*int main() {  
 Student \*student1 = new Student("杜子腾", 39);  
  
 Student \*student2 = student1; // 压根就不会执行拷贝构造函数（指针指向问题，和我们刚刚那个 对象2=对象1 是两回事）  
  
 // 原理，为什么不会？ 纠结  
  
 student2->setAge(99);  
  
 cout << student1->getName() << student1->getAge() << endl;  
  
 // “->”调用一级指针的成员 和 “.”非指针的操作 有啥区别  
  
 return 0;  
}  
  
// Clion和VS差异太大了，放到下一节课，包括 内存地址的分析，等等，到下节课讲  
// 下节课讲： 拷贝构造函数的，深拷贝 和 浅拷贝

### 指针常量 常量指针 常量指针常量

// 指针常量 常量指针 常量指针常量

#include <iostream>  
#include <string.h>  
#include <string.h>  
  
using namespace std;  
  
int main() {  
  
 // \*strcpy (char \*\_\_restrict, const char \*\_\_restrict);  
 // strcpy()  
  
  
 int number = 9;  
 int number2 = 8;  
  
 // 大道至简 一分钟搞定  
  
 // 常量指针  
 const int \* numberP1 = &number;  
 // \*numberP1 = 100; // 报错，不允许去修改【常量指针】存放地址所对应的值  
 // numberP1 = &number2; // OK，允许重新指向【常量指针】存放的地址  
  
 // 指针常量  
 int\* const numberP2 = &number;  
 \*numberP2 = 100; // OK，允许去修改【指针常量】存放地址所对应的值  
 // numberP2 = &number2; // 报错，不允许重新指向【指针常量】存放的地址  
  
 // 常量指针常量  
 const int \* const numberP3 = &number;  
 // \*numberP3 = 100; // 报错，不允许去修改【常量指针常量】存放地址所对应的值  
 // numberP3 = &number2; // 报错，不允许重新指向【常量指针常量】存放的地址  
  
 return 0;  
}

## C++浅拷贝与深拷贝的原理

### 例子1

#define \_CRT\_SECURE\_NO\_WARNINGS // strcpy运行会报错，支持

#include<iostream>  
#include<string.h>  
using namespace std;  
  
class Student1  
{  
public:  
  
 int age;  
 char \* name;  
  
 Student1() { cout << "空参数构造函数" << endl; }  
  
 Student1(char \* name) :Student1(name, 99) { cout << "一个参数构造函数" << endl; }  
  
 Student1(char \* name, int age) {  
 cout << "二个参数构造函数" << endl;  
  
 this->name = (char \*) malloc(sizeof(char \* ) \* 10);  
 strcpy(this->name, name);  
  
 this->age = age;  
 }  
  
 ~Student1() {  
 cout << "析构函数执行" << endl;  
  
 free(this->name);  
 this->name = NULL;  
 }  
  
 // 默认有一个拷贝构造函数 隐士的 我们看不见  
 // Student(const Student & stu) {  
 // stu 旧地址  
  
 // this 新地址  
  
 // s2 = 新地址  
 // }  
};  
  
void mainT1() {  
 // ① 情况分析 画图了  
 // Student s1;  
 // Student s2;  
 // cout << &s1 << endl;  
 // cout << &s2 << endl;  
 // 两个地址 完全不同  
 // 打印：  
 // 空参数构造函数  
 // 空参数构造函数  
 // 1000H  
 // 2000H  
  
  
 // ② 情况分析   
 Student1 s1;  
 Student1 s2 = s1;  
  
 // 两个地址 完全不同  
 // 打印：  
 // 空参数构造函数  
 // 1000H  
 // 2000H  
  
 cout << &s1 << endl;  
 cout << &s2 << endl;  
  
 getchar(); // 不要一闪而过，让程序停留  
}

### 例子2

#define \_CRT\_SECURE\_NO\_WARNINGS // strcpy运行会报错，支持

#include<iostream>  
#include<string.h>  
using namespace std;  
  
class Student2  
{  
public:  
  
 int age;  
 char \* name;  
  
 Student2() { cout << "空参数构造函数" << endl; }  
  
 Student2(char \* name) :Student2(name, 99) {  
 cout << "一个参数构造函数 this:" << this << endl;  
 }  
  
 Student2(char \* name, int age) {  
 cout << "二个参数构造函数 this:" << this << endl;  
  
 this->name = (char \*)malloc(sizeof(char \*)\* 10);  
 strcpy(this->name, name);  
  
 this->age = age;  
 }  
  
 ~Student2() {  
 cout << "析构函数执行 &this->name:" << &this->name << endl;  
  
 free(this->name);  
 this->name = NULL;  
 }  
  
 // 默认有一个拷贝构造函数 隐士的 我们看不见  
 // 一旦复写了拷贝构造函数，默认的还在吗？ Java的构造函数一个思路  
 Student2(const Student2 & stu) {  
 // stu 旧地址  
  
 // this 新地址  
  
 // s2 = 新地址  
  
 cout << "拷贝构造函数 &stu:" << &stu << " this:" << this << endl;  
  
  
 // 新地址name = 旧地址 （浅拷贝）  
 this->name = stu.name;  
  
  
 // 深拷贝 后面见 原理全部打通的时候讲  
 } // 此拷贝构造函数执行完 旧会出现一个 this==新地址 给 main函数的 stu  
};  
  
Student2 getStudent(char \* name) {  
 Student2 stu(name); // 旧地址  
  
 cout << "getStudent函数:" << &stu << endl; // 旧地址  
  
 return stu; // stu 旧地址  
} // 弹栈 释放 栈成员 stu  
  
void mainT3() {  
 // = 会执行拷贝构造函数  
 // stu 新地址  
 Student2 stu = getStudent("截拳道");  
  
 cout << "main函数:" << &stu << endl;  
  
 // 打印：  
 // 两个参数构造函数  
 // 一个参数构造函数  
 // getStudent函数: 1000H地址  
 // 拷贝构造函数 构建新地址 把新地址 给 main函数的 stu == 新地址  
 // 析构函数  
 // main函数:   
  
 // getchar(); // 不要一闪而过，让程序停留  
} // main函数弹栈 stu 新地址 析构函数执行  
  
// 伏笔一： main函数弹栈 stu 新地址 析构函数执行 会造成 重复释放空间的问题 深拷贝  
// 伏笔二 xx

### 例子3

#define \_CRT\_SECURE\_NO\_WARNINGS // strcpy运行会报错，支持

#include<iostream>  
#include<string.h>  
using namespace std;  
  
class Student  
{  
public:  
  
 int age;  
 char \* name;  
  
 Student() { cout << "空参数构造函数" << endl; }  
  
 Student(char \* name) :Student(name, 99) {  
 cout << "一个参数构造函数 this:" << (int)this << endl;  
 }  
  
 Student(char \* name, int age) {  
 cout << "二个参数构造函数 this:" << (int)this << endl;  
  
 this->name = (char \*)malloc(sizeof(char \*)\* 10);  
 strcpy(this->name, name);  
  
 this->age = age;  
 }  
  
 ~Student() {  
 cout << "析构函数执行 &this->name:" << (int)this->name << endl;  
  
 free(this->name);  
 this->name = NULL;  
 }  
  
 // 默认有一个拷贝构造函数 隐士的 我们看不见  
 // 一旦复写了拷贝构造函数，默认的还在吗？ Java的构造函数一个思路  
 // 自定义拷贝构造函数 如果有堆成员，必须采用深拷贝  
 Student(const Student & stu) {  
 // stu 旧地址  
  
 // this 新地址  
  
 // s2 = 新地址  
  
 cout << "拷贝构造函数 &stu:" << (int)&stu << " this:" << (int)this << endl;  
  
 // 【浅拷贝】：新地址name 旧地址name 指向同一个空间，会造成，重复free的问题，引发奔溃  
 // 新地址name = 旧地址 （浅拷贝）  
 // this->name = stu.name;  
  
 // 【深拷贝】  
 this->name = (char \*)malloc(sizeof(char \*)\* 10);  
 strcpy(this->name, name);  
  
 this->age = stu.age;  
  
 cout << "拷贝构造函数2 this->name:" << ((int) this->name) << " stu.name:" << (int)stu.name << endl;  
  
 // 深拷贝 后面见 原理全部打通的时候讲  
 } // 此拷贝构造函数执行完 旧会出现一个 this==新地址 给 main函数的 stu  
  
  
 // 默认的拷贝构造函数 是浅拷贝  
};  
  
void showStudent(Student stu) {  
 cout << "showStudent函数：" << (int)&stu << " " << stu.name << "," << stu.age<< endl;  
}  
  
void main() {  
 Student stu("刘奋", 31);  
  
 showStudent(stu); // 弹栈后 新地址name释放一遍  
 // showStudent(stu); // 弹栈后 新地址name释放一遍  
 // 两次释放新地址name 会奔溃  
  
 // 释放一次新地址name 再释放一次旧name也报错  
  
 showStudent(stu);  
  
  
 showStudent(stu);  
  
  
 showStudent(stu);  
  
  
 showStudent(stu);  
  
 getchar();  
} // main函数弹栈 stu 旧地址  
  
  
// 专业技能：1.研究过C++语言深拷贝原理

## C++语言之this原理与友元函数友元类

### C++ 编写标准

头文件：

#include <iostream>

using namespace std;  
  
// JNI 为什么一定要用这个东西（后面专门）  
  
#ifndef PIG\_H // 你有没有这个宏（Java 宏==常量）  
#define PIG\_H // 定义这个宏  
  
class Pig {  
private:  
 int age;  
 char \* name;  
  
public:  
 // 静态成员声明  
 static int id;  
  
 // 构造函数的声明系列  
 Pig();  
 Pig(char \*);  
 Pig(char \*,int);  
  
 // 析构函数  
 ~Pig();  
  
 // 拷贝构造函数  
 Pig(const Pig & pig);  
  
 // 普通函数 set get  
 int getAge();  
 char \* getName();  
 void setAge(int);  
 void setName(char \*);  
  
 void showPigInfo() const; // 常量指针常量 只读  
  
 // 静态函数的声明  
 static void changeTag(int age);  
  
 // 不要这样干  
 // void changeTag(int age);  
  
 // 友元函数的声明  
 friend void changeAge(Pig \* pig, int age);  
};  
  
#endif // 关闭/结尾

实现文件：

#include "Pig.h"

// *TODO ====================== 下面是 普普通通 常规操作 对象::*// 实现构造函数  
Pig::Pig() {  
 cout << "默认构造函数" << endl;  
}  
  
Pig::Pig(char \* name) {  
 cout << "1个参数构造函数" << endl;  
}  
  
Pig::Pig(char \* name, int age) {  
 cout << "2个参数构造函数" << endl;  
}  
  
// 实现析构函数  
Pig::~Pig() {  
 cout << "析构函数" << endl;  
}  
  
// 实现 拷贝构造函数  
Pig::Pig(const Pig &pig) {  
 cout << "拷贝构造函数" << endl;  
}  
  
int Pig::getAge() {  
 return this->age;  
}  
char \* Pig::getName() {  
 return this->name;  
}  
void Pig::setAge(int age) {  
 this->age = age;  
}  
void Pig::setName(char \* name) {  
 this->name = name;  
}  
  
void Pig::showPigInfo() const {  
  
} // 常量指针常量 只读  
  
  
  
// *TODO =============================== 静态 和 友元 注意点 自己理解*// 实现 静态属性【不需要增加 static关键字】  
int Pig::id = 878;  
  
// 实现静态函数，【不需要增加 static关键字】  
void Pig::changeTag(int age) {  
  
}  
  
// 友元的实现  
// 友元特殊：不需要关键字，也不需要 对象:: ,只需要保证 函数名（参数）  
void changeAge(Pig \* pig, int age) {  
  
}

### C++可变参数。

#include <iostream>

#include <stdarg.h> // 可变参数的支持  
using namespace std;  
  
// Java的可变参数: int ...  
// C++的可变参数写法：...  
// count的第一个用处：内部需要一个 存储地址用的参考值，如果没有第二个参数，内部他无法处理存放参数信息  
void sum(int count, ...) {  
 va\_list vp; // 可变参数的动作  
  
 // 参数一：可变参数开始的动作vp  
 // 参数二：内部需要一个 存储地址用的参考值，如果没有第二个参数，内部他无法处理存放参数信息  
 va\_start(vp, count);  
  
 // 到这里后：vp就已经有丰富的信息  
  
 // 取出可变参数的一个值  
 int number = va\_arg(vp, int);  
 cout << number << endl;  
  
 // 取出可变参数的一个值  
 number = va\_arg(vp, int);  
 cout << number << endl;  
  
 // 取出可变参数的一个值  
 number = va\_arg(vp, int);  
 cout << number << endl;  
  
 // 越界 系统值 乱码  
 // 取出可变参数的一个值 【娶不到后，会取系统值 乱码】  
 number = va\_arg(vp, int);  
 cout << number << endl;  
  
 // 关闭阶段  
 va\_end(vp);  
}  
  
// 1.可变参数  
int main() {  
 std::cout << "同学们大家好，我是Derry" << std::endl;  
  
 sum(546, 6,7,8);  
  
 return 0;  
}

#include <iostream>

#include <stdarg.h> // 可变参数的支持  
using namespace std;  
  
// Java的可变参数: int ...  
// C++的可变参数写法：...  
  
// count变量的第二个用处，用于循环遍历长度  
void sum(int count, ...) {  
 va\_list vp; // 可变参数的动作  
  
 // 参数一：可变参数开始的动作vp  
 // 参数二：内部需要一个 存储地址用的参考值，如果没有第二个参数，内部他无法处理存放参数信息  
 va\_start(vp, count);  
  
 // 到这里后：vp就已经有丰富的信息  
  
 for (int i = 0; i < count; ++i) {  
 int r = va\_arg(vp, int);  
 cout << r << endl;  
 }  
  
 // 关闭阶段（规范：例如：file文件一样 要关闭）  
 va\_end(vp);  
}  
  
// 1.可变参数  
int main() {  
  
 sum(3, 6,7,8); // 真实开发过程的写法  
  
 return 0;  
}

### C++static关键字。

// 2.C++static关键字。 错误的写法

#include <iostream>  
  
using namespace std;  
  
class Dog {  
public:  
 char \* info;  
 int age;  
  
 // 已经编译不成功，不允许这样初始化  
 // static int id = 9;  
 static int id;  
  
 Dog() {  
 // 运行报错  
 // id = 9;  
 }  
  
 static void update() {  
 // 运行报错  
 // id = 9;  
 }  
  
 void update2() {  
 // 运行报错  
 // id = 9;  
 }  
};  
  
int main() {  
 Dog dog;  
 Dog::update(); // 类名::可以调用静态函数  
 return 0;  
}

// 2.C++static关键字。 正确的写法

/\*\*  
 \* 静态的总结：  
 \* 1.可以直接通过类名::静态成员（字段/函数）  
 \* 2.静态的属性必须要初始化，然后再实现（规则）  
 \* 3.静态的函数只能取操作静态的属性和方法（Java）  
 \*/  
  
#include <iostream>  
  
using namespace std;  
  
class Dog {  
public:  
 char \* info;  
 int age;  
  
 // 先声明  
 static int id;  
  
 static void update() {  
 id += 100;  
  
 // 报错:静态函数不能调用非静态函数（Java）  
 // update2();  
 }  
  
 void update2() {  
 id = 13;  
 }  
};  
  
// 再实现  
int Dog::id = 9;  
  
int main() {  
 Dog dog;  
 dog.update2(); // 普通函数  
 Dog::update(); // 静态函数  
 dog.update(); // 对象名.静态函数（一般都是使用::调用静态成员，这种方式可以 知道就行）  
  
 cout << Dog::id << endl;  
 return 0;  
}

### C++对象中，为什么需要this。

// 3.C++对象中，为什么需要 this。

#include <iostream> // iostream.h 早期C++的方式  
  
using namespace std;  
  
class Student {  
private:  
 char \*name;  
 int age;  
  
public:  
 static int id; // 先声明  
  
public:  
 void setName(char \*name) {  
 this->name = name;  
 }  
 void setAge(int age) {  
 this->age = age;  
 }  
 char \*getName() {  
 return this->name;  
 }  
 int getAge() {  
 return this->age;  
 }  
  
public:  
 // 默认的构造函数 栈区开辟空间 暴露 地址 == this指针 (和Java一致的思路)  
};  
  
// 再实现  
int Student::id = 9527;  
  
int main() {  
 // ======= 常规使用下而已  
 Student student;  
 student.setAge(99);  
 student.setName("Derry");  
 cout << student.getName() << " , " << student.getAge()<< endl;  
  
 // ========== this 纠结  
 Student student1;  
 student1.setAge(88); // 设置值的时候，它怎么知道是给student1的age设置值的？  
 student1.id = 880;  
  
 Student student2;  
 student2.setAge(99); // 设置值的时候，它怎么知道是给student2的age设置值的？  
 student2.id = 990;  
  
 Student::id = 666;  
  
 // 它怎么知道是获取student1的age  
 cout << " student1.getAge:" << student1.getAge() << endl;  
  
 // 它怎么知道是获取student2的age  
 cout << " student2.getAge:" << student2.getAge() << endl;  
  
 cout << "student1.id:" << student1.id << endl;  
 cout << "student2.id:" << student2.id << endl;  
 cout << "Student:::" << Student::id << endl;  
  
  
 return 0;  
} // main函数弹栈会 隐士代码：（栈区：delete student ...， 堆区需要自己手动delete）

### const修饰函数的this意义何在。

// 4.const修饰函数的this意义何在。

#include <iostream>  
  
using namespace std;  
  
// 如果是面试C++岗位，会被回到，不然问不到  
  
class Worker {  
public:  
 char \* name;  
 int age = NULL; // C++中不像Java，Java有默认值， 如果你不给默认值，那么就是系统值 -64664  
  
 // int \* const 指针常量 指针常量【地址对应的值能改，地址不可以修改】  
 // const int \* 常量指针 常量指针【地址可以修改，地址对应的值不能改】  
  
 // 纠结：原理：为什么可以修改age  
 // 默认持有隐士的this【类型 \* const this】  
 // 类型 \* const 指针常量：代表指针地址不能被修改，但是指针地址的值是可以修改的  
 void change1() {  
 // 代表指针地址不能被修改  
 // this = 0x6546; // 编译不通过，地址不能被修改，因为是指针常量  
 // 地址不可以修改  
 // this = 0x43563;  
  
 // 隐士的this  
 // 但是指针地址的值是可以修改的  
 // 地址对应的值能改  
 this->age = 100;  
 this->name = "JJJ";  
 }  
  
 // 默认现在：this 等价于 const Student \* const 常量指针常量（地址不能改，地址对应的值不能改）  
 void changeAction() const {  
 // 地址不能改  
 // this = 0x43563;  
  
 // 地址对应的值不能改  
 // this->age = 100;  
 }  
  
 // 原理：修改隐士代码 const 类型 \* const 常量指针常量  
 void showInfo() const {  
 // this->name = "";  
 // this->age = 88;  
  
 // 只读的  
 cout << "age:" << age << endl;  
 }  
};  
  
int main() {  
 return 0;  
}

### 友元函数与友元类实战运用。

// 友元函数

// 老外：你是它的好朋友，那就可以拿私有成员给好朋友  
  
#include <iostream>  
  
using namespace std;  
  
class Person {  
private: // 私有的age，外界不能访问  
 int age = 0;  
  
public:  
 Person(int age) {  
 this->age = age;  
 }  
  
 int getAge() {  
 return this->age;  
 }  
  
 // 定义友元函数 (声明，没有实现)  
 friend void updateAge(Person \* person, int age);  
};  
  
// 友元函数的实现，可以访问所以私有成员  
void updateAge(Person\* person, int age) {  
 // 默认情况下：不能修改 私有的age  
 // 谁有这个权限：友元（拿到所有私有成员）  
 person->age = age;  
}  
  
int main() {  
 Person person = Person(9);  
 updateAge(&person, 88);  
  
 cout << person.getAge() << endl;  
 return 0;  
}

// 友元类 的 小故事 （ImageView 私有成员 可以通过Class来访问，但是Class操作的native

C++代码）  
// 下载 JDK native代码 研究 【同学们自己去研究】  
  
// ImageView 私有成员 你能访问它的私有成员吗 Class  
#include <iostream>

using namespace std;  
  
class ImageView {  
private:  
 int viewSize;  
 friend class Class; // 友元类  
};  
  
// Java每个类，都会有一个Class，此Class可以操作 ImageView私有成员（感觉很神奇）  
class Class {  
public:  
 ImageView imageView;  
  
 void changeViewSize(int size) {  
 imageView.viewSize = size;  
 }  
  
 int getViewSize() {  
 return imageView.viewSize;  
 }  
};  
  
int main() {  
 Class mImageViewClass;  
  
 mImageViewClass.changeViewSize(600);  
  
 cout << mImageViewClass.getViewSize() << endl;  
  
 return 0;  
}

## C++面向对象继承与操作符重载

### 类外运算符重载。

// 1.类外运算符重载。

#include <iostream>  
using namespace std;  
  
class Derry {  
private:  
 int x,y;  
  
public:  
 // 系统C++源码，大量使用此方式 :x(x), y(y)  
 Derry(int x, int y) :x(x), y(y) {}  
  
 // set get 函数  
 void setX(int x) {  
 this->x = x;  
 }  
 void setY(int y) {  
 this->y = y;  
 }  
 int getX() {  
 return this->x;  
 }  
 int getY() {  
 return this->y;  
 }  
};  
  
// 在真实开发过程中，基本上都是写在类的里面的，为什么？ 娓娓道来 答：外部是不能获取内部的私有成员的  
// 把+重载 运算符重载  
Derry operator + (Derry derry1, Derry derry2) {  
 int x = derry1.getX() + derry2.getX();  
 int y = derry1.getY() + derry2.getY();  
  
 Derry res(x, y);  
 return res;  
}  
  
int main() {  
 std::cout << "NDK的第十一节课，同学们大家好" << std::endl;  
  
 // 对象1 + 对象2 C++默认不支持的， Java也不支持，Kotlin也不支持  
  
 // C++/Kotlin 运算符重载 + 把+重载掉  
  
 Derry derry1(1000, 2000);  
 Derry derry2(3000, 4000);  
  
 Derry derry = derry1 + derry2; // C++/Java/Kotlin  
  
 cout << derry.getX() << " , " << derry.getY() << endl;  
  
 return 0;  
}

### 类里运算符重载。

// 2.类里运算符重载。

#include <iostream>  
using namespace std;  
  
class Derry {  
private:  
 int x,y;  
  
public:  
 Derry() {  
  
 }  
  
 // 系统C++源码，大量使用此方式 :x(x), y(y)  
 Derry(int x, int y) :x(x), y(y) {}  
  
 // set get 函数  
 void setX(int x) {  
 this->x = x;  
 }  
 void setY(int y) {  
 this->y = y;  
 }  
 int getX() {  
 // this->x -9; 系统怕你在函数里面 修改了  
 return this->x;  
 }  
 int getY() {  
 return this->y;  
 }  
  
 // +号，运算符重载  
 /\*Derry operator + (Derry derry1) {  
 // this指针 指向当前对象，所以只需要一个  
 int x = this->x + derry1.getX();  
 int y = this->y + derry1.getY();  
 return Derry(x, y);  
 }\*/  
  
 // 系统是这样写的 常量引用：不允许修改，只读模式  
 // const 关键字的解释  
 // & 性能的提高，如果没有& 运行+ 构建新的副本，会浪费性能  
 // 如果增加了& 引用是给这块内存空间取一个别名而已  
 Derry operator + (const Derry& derry1) {  
 // this指针 指向当前对象，所以只需要一个  
 int x = this->x + derry1.x; // 我在类的里面，是可以拿私有成员的  
 int y = this->y + derry1.y; // 我在类的里面，是可以拿私有成员的  
 return Derry(x, y);  
 }  
  
 // 运算符- 重载  
 Derry operator - (const Derry & derry1) {  
 int x = this->x - derry1.x;  
 int y = this->y - derry1.y;  
 return Derry(x, y);  
 }  
  
 // 对象++ 运算符 重载  
 void operator ++() { // ++对象  
 this->x = this->x + 1;  
 this->y = this->y + 1;  
 }  
 void operator ++ (int) { // 对象++  
 this->x = this->x + 1;  
 this->y = this->y + 1;  
 }  
  
 // istream 输入 系统的  
 // ostream 输出 系统的  
 // 输出 运算符重载 复杂 涉及到规则 重载<<  
 friend void operator << (ostream & \_START, Derry derry1) {  
 // 输出换行：<< endl;  
 \_START << " 单 哥开始输出了 " << derry1.x << " ! " << derry1.y << " 哥结束了 " << endl;  
 }  
  
 // 多个的 ostream 输出 系统的  
 // 输出 运算符重载 复杂 涉及到规则 重载 >>  
 /\*friend ostream & operator >> (ostream & \_START, const Derry & derry1) {  
 \_START << " 多 哥开始输出了 " << derry1.x << " ! " << derry1.y << " 哥结束了 " << endl;  
 return \_START;  
  
 // RxJava 链式调用 .操作符.操作符.操作符 你每次都是放回this 一个思路  
 }\*/  
  
 // istream 输入 系统的  
 friend istream & operator >> (istream & \_START, Derry & derry) {  
 // 接收用户的输入，把输入的信息，给x  
  
 // \_START >> derry.x;  
 // \_START >> derry.y;  
  
 // 可读性不好，简化了  
 \_START >> derry.x >> derry.y;  
  
 return \_START;  
 }  
};  
  
int main() {  
 Derry derry1(1000, 2000);  
 Derry derry2(3000, 4000);  
  
 // Derry result = derry1 + derry2;  
  
 // Derry result = derry2 - derry1;  
  
 Derry result(1, 2);  
 result++;  
 ++result;  
  
 cout << result.getX() << " , " << result.getY() << endl;  
  
 cout << endl; // 系统的换行  
  
 // 自定义的，系统没有考虑 你要输出 derry1对象，怎么办？ 我们需要自定义 <<  
 cout << derry1; // 单个的  
  
 // cout >> derry1 >> derry1 >> derry1 >> derry1; // 多个的  
  
 cout << endl; // 系统的换行  
  
 // cout 输出的 C++  
 // cin 输入的 C++  
  
 Derry res;  
 cin >> res; // >> 是我们自己重载的哦  
 cout << "你输入的是：" << res.getX() << endl;  
 cout << "你输入的是：" << res.getY() << endl;  
  
 return 0;  
}

### 括号运算符。

// 3.括号运算符。 数组 系统源码把此括号[i]给重载， 系统重载后的样子 \*(arr+i)

#include <iostream>  
using namespace std;  
  
// 写一个小容器，模拟容器  
class ArrayClass {  
  
private:  
 // C++ 默认都是系统值 size 系统值 -13275  
 int size = 0; // 大小 开发过程中，给size赋默认值，不然会出现，后患无穷的问题  
 int \* arrayValue; // 数组存放 int 类型的很多值  
  
public:  
 void set(int index, int value) {  
 arrayValue[index] = value; // []目前不是我的  
 size+=1;  
 }  
 int getSize() { // size成员的目标：是为了循环可以遍历  
 return this->size;  
 }  
 // 运算符重载 [index]  
 int operator[](int index) {  
 return this->arrayValue[index]; // 系统的  
 }  
};  
  
// 输出容器的内容  
void printfArryClass(ArrayClass arrayClass) {  
 cout << arrayClass.getSize() << endl;  
 for (int i = 0; i < arrayClass.getSize(); ++i) {  
 cout << arrayClass[i] << endl; // []是我们自己的 重载符号  
 }  
}  
  
int main() {  
 // 能在栈区的，尽量在栈区  
 // 1.代码量少  
 // 2.避免麻烦  
 // 3.怕有问题  
 // 4.栈区的回收，不是你负责的，责任推卸  
  
 ArrayClass arrayClass; // 栈区 实例出来的对象，是在堆区了  
  
 arrayClass.set(0, 1000);  
 arrayClass.set(1, 2000);  
 arrayClass.set(2, 3000);  
 arrayClass.set(3, 4000);  
 arrayClass.set(4, 5000);  
  
 printfArryClass(arrayClass);  
  
 return 0;  
}

### C++对象继承。

// 4.C++对象继承。

#include <iostream>  
  
using namespace std;  
  
class Person {  
public:  
 char \*name;  
 int age;  
  
public:  
 Person(char \*name, int age) : name(name) {  
 this->age = age;  
 cout << "Person 构造函数" << endl;  
 }  
  
 void print() {  
 cout << this->name << " , " << this->age << endl;  
 }  
};  
  
// 1.默认是 隐式代码： : private Person  
// 2.私有继承：在子类里面是可以访问父类的成员，但是在类的外面不行  
// 3.必须公开继承，才可以访问父类的成员  
class Student : public Person {  
  
// 类 默认是私有，注意下  
  
private:  
 char \* course;  
  
public:  
 // :父类 , 给自己子类成员初始化  
 Student(char \* name, int age, char\* course) : Person(name, age) , course(course) {  
 cout << "Student 构造函数" << endl;  
 }  
  
 void test() {  
 cout << name << endl;  
 cout << age << endl;  
 print();  
 }  
};  
  
int main() {  
 Student stu("李元霸", 99, "C++");  
  
 // 公开继承，才可以拿父类的成员  
 stu.name = "李四";  
  
 return 0;  
}

### C++多继承二义性

// C++是有多继承的

// Java语言不允许多继承，多继承有歧义，如果Java语言多继承 就会导致代码不健壮，（二义性）  
// Java多实现：做的非常棒，严格避免出现 二义性问题（歧义）  
  
#include <iostream>  
  
using namespace std;  
  
class BaseActivity1 {  
public:  
 void onCreate() {  
 cout << "BaseActivity1 onCreate" << endl;  
 }  
  
 void onStart() {  
 cout << "BaseActivity1 onStart" << endl;  
 }  
  
 void show() {  
 cout << "BaseActivity1 show" << endl;  
 }  
};  
  
class BaseActivity2 {  
public:  
 void onCreate() {  
 cout << "BaseActivity2 onCreate" << endl;  
 }  
  
 void onStart() {  
 cout << "BaseActivity2 onStart" << endl;  
 }  
  
 void show() {  
 cout << "BaseActivity2 show" << endl;  
 }  
};  
  
class BaseActivity3 {  
public:  
 void onCreate() {  
 cout << "BaseActivity3 onCreate" << endl;  
 }  
  
 void onStart() {  
 cout << "BaseActivity3 onStart" << endl;  
 }  
  
 void show() {  
 cout << "BaseActivity3 show" << endl;  
 }  
};  
  
// 子类 继承 三个父类  
class MainActivity1 : public BaseActivity1, public BaseActivity2, public BaseActivity3 {  
public:  
 void onCreate() {  
 cout << "MainActivity1 onCreate" << endl;  
 }  
  
 void onStart() {  
 cout << "MainActivity1 onStart" << endl;  
 }  
  
 void showSonInfo() {  
 cout << "MainActivity1 showSonInfo" << endl;  
 }  
  
 // 解决方案二： 子类上 重写父类的show函数  
 void show() {  
 cout << "MainActivity1 show" << endl;  
 }  
  
};  
  
int main() {  
 // 这个是优先寻找子类的函数，因为特别明确，没有问题，还没有产生歧义（二义性）  
 MainActivity1 mainActivity1; // 子类  
 mainActivity1.onCreate();  
 mainActivity1.onStart();  
 mainActivity1.showSonInfo();  
  
 // error: request for member 'show' is ambiguous  
 // 不明确，二义性，歧义  
 // mainActivity1.show();  
  
 // 解决方案一： 明确指定父类 ::  
 mainActivity1.BaseActivity3::show();  
 mainActivity1.BaseActivity2::show();  
 mainActivity1.BaseActivity1::show();  
  
 // 解决方案二： 子类上 重写父类的show函数  
 mainActivity1.show();  
  
 return 0;  
}

### C++多继承二义性2

// 多继承 二义性2：

// 在真实开发过程中，严格避免出现 二义性  
  
#include <iostream>  
  
using namespace std;  
  
// 祖父类  
class Object {  
public:  
 int number;  
};  
  
// 父类1  
class BaseActivity1 : public Object {  
  
};  
  
// 父类2  
class BaseActivity2 : public Object {  
  
};  
  
// 子类  
class Son : public BaseActivity1, public BaseActivity2 {  
  
 // 第二种解决方案： 在类中定义同名成员，覆盖掉父类的相关成员  
public:  
 int number;  
  
};  
  
  
int main() {  
 Son son;  
  
 // error: request for member 'show' is ambiguous 二义性 歧义  
 // son.number = 2000;  
  
 // 第一种解决方案： :: 明确指定  
 son.BaseActivity1::number = 1000;  
 son.BaseActivity2::number = 1000;  
  
 // 第二种解决方案： 在类中定义同名成员，覆盖掉父类的相关成员  
 son.number = 3000;  
  
 // 第三种解决方案： 【虚基类】 属于 虚继承的范畴  
  
 return 0;  
}

### C++多继承二义性 3

// 第三种解决方案： 【虚基类】 属于 虚继承的范畴

// 真实C++开始，是很少出现，二义性（歧义） 如果出现， 系统源码（系统用 第三种解决方案）  
  
#include <iostream>  
  
using namespace std;  
  
// 祖父类  
class Object{  
public:  
 int number;  
 void show() {  
 cout << "Object show run..." << endl;  
 }  
};  
  
// 等下讲 virtual 的原理是什么 ...  
  
// 父类1  
class BaseActivity1 : virtual public Object {  
// public:int number; // 人为制作二义性 error: request for member 'number' is ambiguous  
};  
  
// 父类2  
class BaseActivity2 : virtual public Object {  
// public:int number;  
};  
  
// 子类  
class Son : public BaseActivity1, public BaseActivity2 {  
  
};  
  
int main() {  
 Object object;  
 BaseActivity1 baseActivity1;  
 BaseActivity2 baseActivity2;  
 Son son;  
  
 object.number = 100;  
 baseActivity1.number = 200;  
 baseActivity2.number = 300;  
 son.number = 400;  
  
 object.show();  
 baseActivity1.show();  
 baseActivity2.show();  
 son.show();  
  
 cout << object.number << endl;  
 cout << baseActivity1.number << endl;  
 cout << baseActivity2.number << endl;  
 cout << son.number << endl;  
  
 return 0;  
}

## C++二义性,多态,纯虚函数,模版函数

1.源码中属性初始化的方式。

// 1.源码中属性初始化的方式。

#include <iostream>  
using namespace std; // 已经声明了  
  
// 人类  
class Person {  
protected:  
 // 注意：string 是 std 命名空间里面的成员，C++源码是这种写法std::string  
 // string内部其实就是对 char\*的封装  
 string name;  
 int age;  
public:  
 Person(string name, int age) :name(name), age(age) {}  
};  
  
// 课程类  
class Course {  
private:  
 string name;  
public:  
 Course(string name) :name(name) {}  
};  
  
class Student : public Person {  
private:  
 // 如果定义的是对象成员，必须这样初始化(构造函数的后面 : 对象成员(内容)) 使用我们的第二种方式  
 Course course; // 对象成员  
public:  
 Student(string name, int age, Course course1, string courseNameInfo)  
 :Person(name, age) // 既然继承了父类就必须给父类的构造函数初始化  
  
 // ,  
  
 // course(course1) // 第二种方式，编译阶段认可的 对象=对象 对象直接的赋值而已  
  
 ,  
  
 course(courseNameInfo) // 第三种方式， 对象(string内容) 直接初始化Course对象 --- 构造函数  
  
 {  
 // this->course = course1; // 第一种方式（对象=对象） 编译阶段不认可，无法监测到你是否真的给course对象成员初始化了  
 }  
};  
  
int main() {  
 Course c("C++");  
 Student student("Derry", 30, c, "NDK内容真多");  
  
 return 0;  
}

2.虚继承，二义性。

// 2.虚继承，二义性。 在开发过程中，不准出现，如果出现，要知道怎么回事

#include <iostream>  
using namespace std; // 已经声明了  
  
// 租父类  
class Object {  
public: string info;  
};  
  
// 父类1/父类2  
class Base1 : virtual public Object {};  
class Base2 : virtual public Object {};  
  
// 子类  
class Main1 : public Base1, public Base2 {};  
  
int main() {  
 Object object; // 在栈区开辟，就会有一个this指针，假设指针是1000H，会有指向的能力  
 Base1 base1; // 在栈区开辟，就会有一个this指针，假设指针是2000H，会有指向的能力  
 Base2 base2; // 在栈区开辟，就会有一个this指针，假设指针是300H，会有指向的能力  
 Main1 main1; // 在栈区开辟，就会有一个this指针，假设指针是4000H，会有指向的能力  
  
 object.info = "A";  
 base1.info = "B";  
 base2.info = "C";  
 main1.info = "D";  
  
 cout << object.info << endl;  
 cout << base1.info << endl;  
 cout << base2.info << endl;  
 cout << main1.info << endl;  
  
  
 exit(0);  
}

3.多态（虚函数）。

// 3.多态（虚函数）。 动态多态（程序的角度上：程序在运行期间才能确定调用哪个类的函数 == 动态多态的范畴）  
  
// Java语言默认支持多态  
// C++默认关闭多态，怎么开启多态？ 虚函数 在父类上给函数增加 virtual关键字  
  
#include <iostream>  
using namespace std;  
  
// Android标准  
class BaseActivity {  
public:  
 virtual void onStart() {  
 cout << "BaseActivity onStart" << endl;  
 }  
};  
  
class HomeActivity : public BaseActivity {  
public:  
 void onStart() { // 重写父类的函数  
 cout << "HomeActivity onStart" << endl;  
 }  
};  
  
class LoginActivity : public BaseActivity {  
public:  
 void onStart() { // 重写父类的函数  
 cout << "LoginActivity onStart" << endl;  
 }  
};  
  
// 在此函数 体系多态，例如：你传入HomeActivity，我就帮你运行HomeActivity  
void startToActivity(BaseActivity \* baseActivity) {  
 baseActivity->onStart();  
}  
  
int main() {  
 // *TODO 第一版本* HomeActivity \*homeActivity = new HomeActivity();  
 LoginActivity \*loginActivity = new LoginActivity();  
  
 startToActivity(homeActivity);  
 startToActivity(loginActivity);  
  
 if (homeActivity && loginActivity) delete homeActivity; delete loginActivity;  
  
  
 cout << endl;  
  
  
 // *TODO 第二个版本* BaseActivity \* activity1 = new HomeActivity();  
 BaseActivity \* activity2 = new LoginActivity();  
 startToActivity(activity1);  
 startToActivity(activity2);  
  
  
 // *TODO 抛开 C++ 抛开Java 等等，请问什么是多态？ 父类的引用指向之类的对象，同一个方法有不同的实现，重写（动态多态）和 重载(静态多态)* return 0;  
}

4.静态多态。

// 静态多态 （编译期已经决定，调用哪个函数了，这个就属于静态多态的范畴） 重载（静态多态）  
  
#include <iostream>  
  
using namespace std;  
  
void add(int number1, int number2) {  
 cout << number1 + number2 << endl;  
}  
  
void add(float number1, float number2) {  
 cout << number1 + number2 << endl;  
}  
  
void add(double number1, double number2) {  
 cout << number1 + number2 << endl;  
}  
  
int main() {  
 add(10000, 10000);  
 add(1.9f, 2.8f);  
 add(545.4, 654.54);  
  
 return 0;  
}

5.纯虚函数（Java版抽象类）

// 5.纯虚函数（Java版抽象类）  
// C++纯虚函数(C++没有抽象类) 相当于 Java的抽象类 为了更好理解  
  
#include <iostream>  
using namespace std;  
  
// 抽象类/纯虚函数： 分为：1.普通函数， 2.抽象函数/纯虚函数  
class BaseActivity {  
private:  
 void setContentView(string layoutResID) {  
 cout << "XmlResourceParser解析布局文件信息... 反射" << endl;  
 }  
  
public:  
 // 1.普通函数  
 void onCreate() {  
 setContentView(getLayoutID());  
  
 initView();  
 initData();  
 initListener();  
 }  
  
 // 纯虚函数是必须继承的（如果子类没有重写纯虚函数，子类就是抽象类）， 虚函数是不是不必须的  
  
 // 2.抽象函数/纯虚函数  
 // virtual string getLayoutID(); // 虚函数  
 virtual string getLayoutID() = 0; // 纯虚函数  
 virtual void initView() = 0;  
 virtual void initData() = 0;  
 virtual void initListener() = 0;  
};  
  
// 子类 MainActivity  
class MainActivity : public BaseActivity { // MainActivity如果没有重新父类的纯虚函数，自己就相当于 抽象类了  
  
 string getLayoutID() {  
 return "R.layout.activity\_main";  
 }  
  
 void initView() {  
 // Button btLogin = findViewById(R.id.bt\_login);  
 // Button btRegister = findViewById(R.id.bt\_register);  
 // TextView tvInfo = findViewById(R.id.tv\_info);  
 // ... 省略  
 }  
  
 void initData() {  
 // tvInfo.setText("info...");  
 // ... 省略  
 }  
  
 void initListener() {  
 /\*btLogin.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View v) {  
 // 点击做事情  
 }  
 });\*/  
 // ... 省略  
 }  
};  
  
// 子类 HomeActivity  
class HomeActivity : public BaseActivity { // MainActivity如果没有重新父类的纯虚函数，自己就相当于 抽象类了  
  
 string getLayoutID() {  
 return "R.layout.activity\_home";  
 }  
  
 void initView() {  
 // Button btLogin = findViewById(R.id.bt\_login);  
 // Button btRegister = findViewById(R.id.bt\_register);  
 // TextView tvInfo = findViewById(R.id.tv\_info);  
 // ... 省略  
 }  
  
 void initData() {  
 // tvInfo.setText("info...");  
 // ... 省略  
 }  
  
 void initListener() {  
 /\*btLogin.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View v) {  
 // 点击做事情  
 }  
 });\*/  
 // ... 省略  
 }  
};  
  
// 子类 LoginActivity  
class LoginActivity : public BaseActivity { // MainActivity如果没有重新父类的纯虚函数，自己就相当于 抽象类了  
  
 string getLayoutID() {  
 return "R.layout.activity\_login";  
 }  
  
 void initView() {  
 // Button btLogin = findViewById(R.id.bt\_login);  
 // Button btRegister = findViewById(R.id.bt\_register);  
 // TextView tvInfo = findViewById(R.id.tv\_info);  
 // ... 省略  
 }  
  
 void initData() {  
 // tvInfo.setText("info...");  
 // ... 省略  
 }  
  
 void initListener() {  
 /\*btLogin.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View v) {  
 // 点击做事情  
 }  
 });\*/  
 // ... 省略  
 }  
};  
  
int main() {  
  
 // 错误：抽象类型 MainActivity 绝对不能实例化  
 // MainActivity mainActivity;  
  
 // 重新了父类所有的纯虚函数  
 MainActivity mainActivity;  
  
 return 0;  
}

6.全纯虚函数（Java版接口）

// 虚函数 纯虚函数 全纯虚函数（C++没有接口） 等价于 6.全纯虚函数（Java版接口）。  
  
#include <iostream>  
using namespace std;  
  
class Student {  
 int \_id;  
 string name;  
 int age;  
};  
  
// 此类所有的函数 ，都是纯虚函数，就相当于 Java的接口了  
class ISudent\_DB {  
 virtual void insertStudent(Student student) = 0;  
 virtual void deleteStudent(int \_id) = 0;  
 virtual void updateStudent(int \_id, Student student) = 0;  
 virtual Student queryByStudent(Student student) = 0;  
};  
  
// Java的实现类  
class Student\_DBImpl1 : public ISudent\_DB {  
public:  
 void insertStudent(Student student) {  
 // 插入操作，省略代码...  
 }  
  
 void deleteStudent(int \_id) {  
 // 删除操作，省略代码...  
 }  
  
 void updateStudent(int \_id, Student student) {  
 // 更新操作，省略代码...  
 }  
  
 Student queryByStudent(Student student) {  
 // 查询操作，省略代码...  
 }  
};  
  
// Java的实现类  
class Student\_DBImpl2 : public ISudent\_DB {  
public:  
 void insertStudent(Student student) {  
 // 插入操作，省略代码...  
 }  
  
 void deleteStudent(int \_id) {  
 // 删除操作，省略代码...  
 }  
  
 void updateStudent(int \_id, Student student) {  
 // 更新操作，省略代码...  
 }  
  
 Student queryByStudent(Student student) {  
 // 查询操作，省略代码...  
 }  
};  
  
// Java的实现类  
class Student\_DBImpl3 : public ISudent\_DB {  
public:  
 void insertStudent(Student student) {  
 // 插入操作，省略代码...  
 }  
  
 void deleteStudent(int \_id) {  
 // 删除操作，省略代码...  
 }  
  
 void updateStudent(int \_id, Student student) {  
 // 更新操作，省略代码...  
 }  
  
 Student queryByStudent(Student student) {  
 // 查询操作，省略代码...  
 }  
};  
  
int main() {  
 Student\_DBImpl1 studentDbImpl1;  
 Student\_DBImpl2 studentDbImpl2;  
 Student\_DBImpl3 studentDbImpl3;  
  
 cout << "Success" << endl;  
  
 return 0;  
}

7.回调

// 7.回调。 Java的登录 简单的 接口  
  
#include <iostream>  
using namespace std;  
  
// 登录成功的Bean  
class SuccessBean {  
public:  
 string username;  
 string userpwd;  
  
 SuccessBean(string username, string userpwd)  
 :username(username), userpwd(userpwd) {}  
};  
  
// 登录响应的接口 成功，错误  
class ILoginResponse {  
public:  
 // 登录成功  
 virtual void loginSuccess(int code, string message, SuccessBean successBean) = 0;  
  
 // 登录失败  
 virtual void loginError(int code, string message) = 0;  
};  
  
// 登录的API动作  
void loginAction(string name, string pwd, ILoginResponse & loginResponse) {  
 if (name.empty() || pwd.empty()) {  
 cout << "用户名或密码为空!" << endl;  
 return;  
 }  
  
 if ("Derry" == name && "123" == pwd) {  
 loginResponse.loginSuccess(200, "登录成功", SuccessBean(name, "恭喜你进入"));  
 } else {  
 loginResponse.loginError(404, "登录错误，用户名或密码错误...");  
 }  
}  
  
// 写一个实现类，继承接口  
// 接口实现类  
class ILoginResponseImpl : public ILoginResponse {  
public:  
 // 登录成功  
 void loginSuccess(int code, string message, SuccessBean successBean) {  
 cout << "恭喜登录成功 " << "code:" << code << " message:" << message  
 << "successBean:" << successBean.username << "," << successBean.userpwd << endl;  
 }  
  
 // 登录失败  
 void loginError(int code, string message) {  
 cout << " 登录失败 " << "code:" << code << " message:" << message << endl;  
 }  
};  
  
int main() {  
  
 // 做实验  
 // Allocating an object of abstract class type 'ILoginResponse'  
 // 正在分配抽象类型为ILoginResponse的对象 不能被实例化  
 // 纠结：为什么不可以  
 // 1.他不是Java的接口，C++也没有接口，他只是像接口而已。  
 // 2.他也不是抽象类，C++也没有抽象类，他只是像抽象类而已。  
 // 3.他是纯虚函数的类，此类决定不准你实例化 无论堆区 还是栈区  
 /\*new ILoginResponse() {  
 // 登录成功  
 void loginSuccess(int code, string message, SuccessBean successBean) {  
  
 }  
  
 // 登录失败  
 void loginError(int code, string message) {  
  
 }  
 }\*/  
  
 string username;  
 cout << "请输入用户名.." << endl;  
 cin >> username;  
  
 string userpwd;  
 cout << "请输入密码.." << endl;  
 cin >> userpwd;  
  
 ILoginResponseImpl iLoginResponse;  
 loginAction(username, userpwd, iLoginResponse);  
  
 return 0;  
}

8.模版函数（Java版泛型）

// 8.模版函数（Java版泛型）。 C++没有泛型 C++的模板函数 非常类似于 Java的泛型  
  
#include <iostream>  
  
using namespace std;  
  
// 加分合集 int double float ... 你都要考虑，你是不是要定义很多的 函数  
/\*void addAction(int n1, int n2) {  
 cout << "addAction(int n1, int n2):" << n1 + n1 << endl;  
}  
  
void addAction(float n1, float n2) {  
 cout << "addAction(int n1, int n2):" << n1 + n1 << endl;  
}  
  
void addAction(double n1, double n2) {  
 cout << "addAction(int n1, int n2):" << n1 + n1 << endl;  
}\*/  
  
// 模板函数 == Java的泛型解决此问题  
template <typename TT>  
void addAction(TT n1, TT n2) {  
 cout << "模板函数：" << n1 + n2 << endl;  
}  
  
int main() {  
 addAction(1, 2);  
 addAction(10.2f, 20.3f);  
 addAction(545.34, 324.3);  
 addAction<string>("AAA", "BBB");  
  
 /\*addAction(2, 324.3);  
 addAction(54, 324.3f);\*/  
 return 0;  
}

补充点： 继承关系的时候，构造函数和析构函数 的顺序问题

// 补充点： 继承关系的时候，构造函数和析构函数 的顺序问题  
  
#include <iostream>  
  
using namespace std;  
  
class Person {  
public:  
 string name;  
  
 Person(string name) : name(name) {cout << "Person构造函数" << endl;}  
  
 ~Person() {cout << "Person析构函数" << endl;}  
  
 virtual void test() {  
 cout << "父 test..." << endl;  
 }  
};  
  
class Student : public Person {  
public:  
 string name;  
  
 Student(string name) : Person(name) {  
 cout << "Student构造函数" << endl;  
  
 // Person::test();  
 }  
  
 ~Student() {cout << "Student析构函数" << endl;}  
  
 void test() {  
 cout << "子 test..." << endl;  
 }  
};  
  
int main() {  
 Student student("Derry");  
 // Person构造函数  
 // Student构造函数  
 // Student析构函数  
 // Person析构函数  
  
 Student student1("A");  
 student1.test();  
  
 return 0;  
}

## C++高级之SLT中的容器学习与函数谓词

1.vector向量学习

// Java的集合框架 非常强大 相当于 C++ STL（标准模板库） #include <iostream>  
  
// *TODO STL 是“Standard Template Library”的缩写，中文译为“标准模板库”。STL 是 C++ 标准库的一部分，不用单独安装。*// STL 有很多很多的容器  
// C++ STL 设计思路 和 Java不同  
  
// C ++ vector 向量 容器 （内部：封装动态大小数组作为容器，能够存放任意的动态数组【数据结构】）  
  
#include <iostream>  
  
#include <vector> // 引入 vector 容器的支持  
  
// NDK 开发 一定要用容器 应该 queue 队列  
  
using namespace std;  
  
int main() {  
 std::cout << "标准模板库" << std::endl;  
  
 vector<int> vector1;  
  
 vector<int> vector2(10); // 指定10的空间大小  
  
 vector<int> vector3(10, 0); // 有了10个值了 每个值都是0  
  
 vector<int> vector4;  
  
 // vector4.begin() 迭代器 插入到前面  
 // vector4.end() 迭代器 插入到后面  
  
 // 插入数据  
 vector4.insert(vector4.begin(), 40);  
 vector4.insert(vector4.begin(), 60);  
 vector4.insert(vector4.begin(), 80);  
 vector4.insert(vector4.begin(), 100);  
 vector4.insert(vector4.begin(), 200);  
  
 // 第一个  
 cout << " 修改前：vector4.front():" << vector4.front() << endl;  
 vector4.front() = 99; // 默认修改第一个  
 cout << " 修改后：vector4.front():" << vector4.front() << endl;  
  
 // 最后一个  
 cout << " 修改前：vector4.back():" << vector4.back() << endl;  
 vector4.back() = 777; // 默认修改最后  
 cout << " 修改后：vector4.back():" << vector4.back() << endl;  
  
 vector4.erase(vector4.begin()); // 移除第一个元素（内部：通过迭代器的位置 进行移除） 删除  
  
 // 循环打印，默认 从大到小输出  
 for (int i = 0; i < vector4.size(); ++i) {  
 cout << "item:" << vector4[i] << endl;  
 }  
  
 // KT的类型推到 相当于 C++ auto  
 /\*var a = 10;  
 var b = 30.4f;  
 var c = "";\*/  
  
 // 迭代器 循环遍历  
 // auto Kotlin自带类型推到  
 // for (vector<int>::iterator iteratorVar = vector4.begin(); iteratorVar != vector4.end(); iteratorVar++) {  
 for (auto iteratorVar = vector4.begin(); iteratorVar != vector4.end(); iteratorVar++) {  
 // 迭代器 当中指针操作 iteratorVar++  
 cout << "迭代器：" << \*iteratorVar << endl;  
 }  
  
 cout << "" << endl;  
  
 return 0;  
}

2.stack栈学习

// 2.stack栈学习。 先进后出，后进先出，手枪子弹上膛的一个思路。  
  
#include <iostream>  
#include <stack>  
using namespace std;  
  
// NDK开发者 几乎用不到  
  
int main() {  
 stack<int> stackVar;  
  
 // 压栈（注意：stack无法指定那个位置去压栈）  
 stackVar.push(30);  
 stackVar.push(60);  
 stackVar.push(90);  
  
 // 请问那个元素，在最上面（看图）  
  
 // [] 角标是不行的，内部没有重载此运算符  
 /\*for (int i = 0; i < stackVar.size(); ++i) {  
 // cout << stackVar[i] << endl;  
 // cout << stackVar.at() << endl;  
 }\*/  
  
 // 他根本就没有迭代器  
 // 开发者 自己组装 迭代器 遍历  
 /\*for (stack<int>::iterator; i < ; ++i) {  
  
 }\*/  
  
 // 慎用，为什么？ 元素被弹完了  
 // 这种方式是可以的，手枪把子弹全部打完【会把元素清空】  
 while (!stackVar.empty()) {  
 int top = stackVar.top(); // top == 获取栈顶的元素  
 cout << "获取栈顶的元素：" << top << endl; // 永远拿 90  
  
 stackVar.pop(); // 把栈顶的元素 弹出去 【删除】  
 }  
  
 /\*int top = stackVar.top();  
 cout << top << endl;\*/  
  
 return 0;  
}

3.queue队列学习（必须会使用）

// 3.queue队列学习（必须会使用）。  
  
#include <iostream>  
#include <queue> // 队列支持（内部：基本上 链表 、 数组 ）  
  
// 容器 + 数据结构  
  
// 音视频： 压缩格式音频数据（push） 保存 pop消费最前面的 音频格式数据 获取  
  
using namespace std;  
  
int main() {  
 queue<int> queueVar;  
  
 queueVar.push(20);  
 queueVar.push(40);  
 queueVar.push(60);  
  
 // 第一个元素是谁 20 FIFO 原则  
 cout << " 修改前： queueVar.front():" << queueVar.front() << endl;  
 queueVar.front() = 88;  
 cout << " 修改后： queueVar.front():" << queueVar.front() << endl;  
  
 // 最后一个  
 cout << " 修改前： queueVar.back():" << queueVar.back() << endl;  
 queueVar.back() = 88;  
 cout << " 修改后： queueVar.back():" << queueVar.back() << endl;  
  
 // 没有找到 角标  
 /\*for (int i = 0; i < 9; ++i) {  
 queueVar[i];  
 }\*/  
  
 // 他根本就没有迭代器  
 /\*for (queue<int>::iterator; i < ; ++i) {  
  
 }\*/  
  
 // 慎用，为什么？ 前面的元素全部被消费完了  
 while (!queueVar.empty()) {  
 cout << "while1:" << queueVar.front() << endl;  
  
 queueVar.pop(); // 把前面的元素 给消费掉 【删除】  
 }  
  
 while (!queueVar.empty()) {  
 cout << "while2:" << queueVar.front() << endl;  
  
 queueVar.pop(); // 把前面的元素 给消费掉 【删除】  
 }  
  
 return 0;  
}

4.优先级队列学习

// 4.优先级队列学习 (queue 的 子集)。  
  
#include <iostream>  
#include <queue>  
using namespace std;  
  
// priority\_queue 内部对我们前面的vector 有一定的封装  
  
// Deque 跟 queue 有什么区别啊 双端队列  
  
int main() {  
 // 优先级队列，默认做了 从大大小 60 50 40 ...  
 // priority\_queue<int> priorityQueue;  
  
 // 隐式代码： vector<int>, less<int> 你看不到  
 // priority\_queue<int, vector<int>, less<int>> priorityQueue;  
  
 // less // less<int> return \_\_x < \_\_y; 上一个元素和当前元素 比较，返回ture false 从大到小  
  
 // less return \_\_x < \_\_y; // 从大到小  
 // greater return \_\_x > \_\_y; // 从小到大  
  
 // 第二个参数：他内部需要vector，你就给他  
  
 priority\_queue<int ,vector<int>, less<int>> priorityQueue;  
 // priority\_queue<int ,vector<int>, greater<int>> priorityQueue;  
  
 priorityQueue.push(10);  
 priorityQueue.push(20);  
 priorityQueue.push(30);  
 priorityQueue.push(40);  
 priorityQueue.push(50);  
 priorityQueue.push(60);  
  
 cout << priorityQueue.top() << endl; // 60  
  
 // 循环代码  
 while (!priorityQueue.empty()) {  
 cout << "while1:" << priorityQueue.top() << endl; // 一直是60  
 priorityQueue.pop(); // 最前面的元素消费掉  
 }  
 return 0;  
}

5.list的学习

// 5.list的学习。 Java：ArrayList采用Object[]数组， C++的list 内部：采用链表  
  
#include <iostream>  
#include <list> // list容器的支持  
using namespace std;  
  
// NDK 基本上用不到 list  
  
int main() {  
 list<int> listVar;  
  
 // 插入操作  
 listVar.push\_front(50); // 插入到前面 明确  
 listVar.push\_back(60); // 插入到后面  
 listVar.insert(listVar.begin(), 70); // 插入到前面 灵活  
 listVar.insert(listVar.end(), 80); // 插入到后面  
  
 // 修改操作  
 listVar.back() = 88;  
 listVar.front() = 55;  
  
 // 删除  
 listVar.erase(listVar.begin()); // 删除最前面的 55  
 listVar.erase(listVar.end()); // 删除最后面的 88  
  
 // list 迭代器  
 // 不用通过角标去访问，也不能修改 遍历  
 for (list<int>::iterator it = listVar.begin(); it != listVar.end() ; it ++) {  
 cout << \*it << endl;  
 }  
  
 return 0;  
}

6.set来引出函数谓词。

// 6.set容器 来引出函数谓词。  
  
// set（内部：红黑树结构），会对你存入的数据进行排序，但是绝对不允许元素相同  
  
#include <iostream>  
#include <set>  
  
using namespace std;  
  
  
int main() {  
 set<int, less<int>> setVar; // \_\_x < \_\_y 从小到大，默认情况下 就是 less  
  
 // 添加参数，不需要用迭代器，也不需要指定位置  
 setVar.insert(1);  
 setVar.insert(3);  
 setVar.insert(2);  
 setVar.insert(4);  
  
 // 重复插入，并不会报错 std::pair<iterator, bool>  
 pair<set<int, less<int>>::iterator, bool> res = setVar.insert(8);  
  
 // res.first 获取第一个元素 迭代器 当前迭代器 最后一个位置  
 // res.second 获取第二个元素 bool  
 bool insert\_success = res.second;  
 if (insert\_success) {  
 cout << "恭喜你，插入成功" << endl;  
  
 // 插入成功后，我用第一个元素遍历  
 for (; res.first != setVar.end(); res.first ++) {  
 cout << \*res.first << endl;  
 }  
  
 } else {  
 cout << "哎，插入失败.." << endl;  
 }  
  
 // 全部遍历 auto 自动推到  
 for (auto it = setVar.begin(); it != setVar.end() ; it ++) {  
 cout << \*it << endl;  
 }  
  
 return 0;  
}

谓词 设计对象的排序

// 谓词 设计对象的排序  
  
#include <iostream>  
#include <set>  
  
using namespace std;  
  
// set（内部：红黑树结构），会对你存入的数据自动进行排序，但是绝对不允许元素相同  
  
class Person {  
public:  
 string name;  
 int id;  
 Person(string name, int id) : name(name), id(id) {}  
};  
  
// C++ 都是一些常规的 谓词 不能满足功能， 模仿C++源码的谓词 自定义谓词 解决这个问题  
  
// C++缔造者写的这个源码 没有对象比较的功能 【系统源码谓词 做不到对象比较功能】  
// bool operator()(const \_Tp& \_\_x, const \_Tp& \_\_y) const { return \_\_x < \_\_y; }  
  
// 我们就自定义这个功能 【自定义谓词 没有达到谓词的标准】  
bool doCompareAction(const Person& person1, const Person& person2) {  
 return person1.id < person2.id;  
};  
  
// 真正的谓词  
struct doCompareAction2 {  
public:  
 bool operator() (const Person& \_\_x, const Person& \_\_y) {  
 return \_\_x.id < \_\_y.id;  
 }  
};  
  
struct doCompareAction3 {  
public:  
 bool operator() (const Person& \_\_x, const Person& \_\_y) {  
 return \_\_x.id > \_\_y.id;  
 }  
};  
  
int main() {  
 // 默认是 less return 对象1 < 对象2;  
  
 set<Person, doCompareAction2> setVar;  
  
 // set<Person, doCompareAction3> setVar;  
  
 // 构建对象  
 Person p1 ("Snake", 1);  
 Person p2 ("kevin", 2);  
 Person p3 ("Derry", 3);  
  
 // 把构建的对象 插入到 set 容器里面去  
 setVar.insert(p1);  
 setVar.insert(p2);  
 setVar.insert(p3);  
  
 // name string --- c\_str() ----> char \*  
 for (set<Person>::iterator it = setVar.begin(); it != setVar.end() ; it ++) {  
 cout << it->name.c\_str() << " , " << it->id << endl;  
 }  
  
 return 0;  
}

## C++高级之C++内部源码阅读与手写多元谓词

1.map容器学习。

#include <iostream>

#include <map>  
  
using namespace std;  
  
int main() {  
 cout << "1.map容器学习。" << endl;  
  
 // 注意：map会对key进行排序，默认 key不能重复  
 map<int, string> mapVar;  
  
 // *TODO 添加数据* // 第一种方式  
 mapVar.insert(pair<int, string>(1, "一"));  
  
 // 第二种方式  
 mapVar.insert(make\_pair(2, "二"));  
  
 // 第三种方式  
 mapVar.insert(map<int, string>::value\_type (3, "三"));  
  
 // 上面三种方式 key不能重复  
 // 思考：既然会对key进行排序，那么key是不能重复的（会插入失败）  
 mapVar.insert(pair<int, string>(3, "三3"));  
  
 // 第四种方式 mapVar[key]=Value  
 mapVar[4] = "四";  
 mapVar[4] = "肆"; // 第四种方式覆盖/替换（常用）  
  
 /\*\*  
 \* typedef typename \_Rep\_type::iterator iterator; 之前常规的迭代器  
 typedef typename \_Rep\_type::const\_iterator const\_iterator; 只读的，只能读，不能修改 的迭代器  
 typedef typename \_Rep\_type::reverse\_iterator reverse\_iterator; 倒序打印的迭代器  
 \*/  
  
 // 循环打印，迭代器  
 for (map<int, string>::iterator it = mapVar.begin() ; it != mapVar.end() ; it ++) {  
 cout << it->first << "," << it->second.c\_str() << "\t";  
 }  
 cout << endl;  
  
 // Derry想知道，怎么判断插入成功、失败  
 std::pair<map<int, string>::iterator, bool> result = mapVar.insert(map<int, string>::value\_type (6, "66三san"));  
 if (result.second) {  
 cout << "插入成功" << endl;  
 } else {  
 cout << "插入失败" << endl;  
 }  
 // 我想知道插入后的 数据  
 for (result.first == mapVar.begin(); result.first != mapVar.end() ; result.first++) {  
 cout << result.first->first << " , " << result.first->second << endl;  
 }  
  
 // 查找，操作  
 map<int, string> ::iterator findResult = mapVar.find(3); // 查找  
 if (findResult != mapVar.end()) {  
 cout << "恭喜，找到了" << findResult->first << "," << findResult->second.c\_str() << endl;  
 } else {  
 cout << "不恭喜，没找到了" << endl;  
 }  
  
 return 0;  
}

2.multimap容器学习。

// 2.multimap容器学习。

#include <iostream>  
#include <map>  
  
using namespace std;  
  
int main() {  
 // 1.key可以重复， 2.key重复的数据可以分组, 3.key会排序， 4.value不会排序  
 multimap<int, string> multimapVar;  
  
 multimapVar.insert(make\_pair(10, "十个1"));  
 multimapVar.insert(make\_pair(10, "十个2"));  
 multimapVar.insert(make\_pair(10, "十个3"));  
  
 multimapVar.insert(make\_pair(30, "三十1"));  
 multimapVar.insert(make\_pair(30, "三十3"));  
 multimapVar.insert(make\_pair(30, "三十2"));  
  
  
 multimapVar.insert(make\_pair(20, "二十1"));  
 multimapVar.insert(make\_pair(20, "二十2"));  
 multimapVar.insert(make\_pair(20, "二十3"));  
  
 for (auto iteratorVar = multimapVar.begin(); iteratorVar != multimapVar.end() ; iteratorVar ++) {  
 cout << iteratorVar->first << "," << iteratorVar->second << endl;  
 }  
 cout << endl;  
  
 // *TODO 核心功能是分组* int result;  
 cout << "请输入你要查询的key，为int类型:" << endl;  
 cin >> result;  
  
 multimap<int, string>::iterator iteratorVar = multimapVar.find(result);  
 while (iteratorVar != multimapVar.end()) {  
 cout << iteratorVar->first << "," << iteratorVar->second << endl;  
  
 // 需要自己做逻辑控制，不然有问题  
 iteratorVar++;  
  
 if (iteratorVar->first != result) {  
 break;; // 循环结束  
 }  
  
 // 严谨性  
 if (iteratorVar == multimapVar.end()) {  
 break;; // 循环结束  
 }  
 }  
  
 return 0;  
}

3.C++内部源码的仿函数。

4.多元谓词仿函数回顾操练。

// 谓词前戏  
  
#include <iostream>  
#include <set> // STL包  
#include <algorithm> // 算法包  
  
using namespace std;  
  
// 谓词 == 仿函数  
  
// *TODO 第一版: 为什么叫仿函数 （空谓词 一元谓词 二元谓词 三元谓词）*class ComPareObject {  
public:  
 void operator()() { // 重装了括号运算符  
 cout << "仿函数" << endl;  
 }  
};  
  
// 普通函数  
void fun2() {  
 cout << "普通函数" << endl;  
}  
  
int main() {  
 ComPareObject fun1;  
  
 // 非常像函数的调用，很像函数 = （仿函数）  
 fun1();  
  
 fun2();  
  
 return 0;  
}

// 谓词中戏  
  
#include <iostream>  
#include <set> // STL包  
#include <algorithm> // 算法包  
  
using namespace std;  
  
// 我如何阅读C++源码，来写我们的仿函数  
// 明明白白的仿函数（一元谓词==一元函数对象）  
class showActionObj {  
public:  
 void operator()(int content) {  
 cout << "自定义仿函数" << content << endl;  
 }  
};  
  
// 回调函数 如果叫 仿函数 有点合理  
// 简洁方式（回调函数、一元谓词 但是不能称为 仿函数）  
void showAction(int content) {  
 cout << "自定义 一元谓词" << content << endl;  
}  
  
using namespace std;  
  
int main() {  
 set<int> setVar;  
  
 setVar.insert(10);  
 setVar.insert(20);  
 setVar.insert(30);  
 setVar.insert(40);  
 setVar.insert(50);  
 setVar.insert(60);  
  
 // for\_each(setVar.begin(), setVar.end(), showActionObj());  
  
 for\_each(setVar.begin(), setVar.end(), showAction);  
  
 return 0;  
}

// 回调函数 和 仿函数 各个的用途 同学们知道了 1  
// 谓词尾戏  
  
#include <iostream>  
#include <set> // STL包  
#include <algorithm> // 算法包  
  
using namespace std;  
  
// 回调函数 (功能够简单)  
void showAction(int \_\_first) {  
 cout << "一元谓词" << \_\_first << endl;  
}  
  
// 仿函数(扩展性强) C++内置源码使用仿函数频率高，扩展性强  
class showActionObj {  
public:  
 int count = 0;  
 void \_count() { cout << "本次输出次数是:" << this->count << endl; }  
  
 void operator() (int \_\_first) {  
 cout << "仿函数" << \_\_first << endl;  
 count++;  
 }  
};  
  
int main() {  
 // 理解：类型传递  
 // set<int, showActionObj> setVar; 这样写的语法是OK的，不能加括号  
 set<int> setVar;  
  
 setVar.insert(10);  
 setVar.insert(20);  
 setVar.insert(30);  
 setVar.insert(40);  
 setVar.insert(50);  
 setVar.insert(60);  
  
 // *TODO 第一种方式* for\_each(setVar.begin(), setVar.end(), showAction);  
 // 请你统计打印次数？ 答：做不到  
  
 // *TODO 第二种方式* showActionObj s; // 理解：值传递  
 s = for\_each(setVar.begin(), setVar.end(), s); // 传入进去的s是新的副本，我们外面的s是旧地址  
 // 请你统计打印次数？ 答：OK  
 s.\_count();  
  
 return 0;  
}

// 类型传递 的 仿函数 怎么看源码得知 写法 同学们看懂了吗  
  
#include <iostream>  
#include <set>  
using namespace std;  
  
// C++源码：typename \_Compare = std::less less内置的仿函数，根据内置仿函数去写 自定义  
// bool operator()(const \_Tp& \_\_x, const \_Tp& \_\_y) const 二元谓词  
class CompareObjectClass {  
public:  
 bool operator() (const string & \_\_x, const string & \_\_y) const { // const 指针 const 常量指针常量 = 只读  
 return \_\_x > \_\_y;  
 }  
};  
  
int main() {  
 set<string, CompareObjectClass> setVar; // 第一版  
  
 setVar.insert(setVar.begin(), "AAAAAAA");  
 setVar.insert(setVar.begin(), "BBBBBBB");  
 setVar.insert(setVar.begin(), "CCCCCCC");  
 setVar.insert(setVar.begin(), "DDDDDDD");  
 setVar.insert(setVar.begin(), "EEEEEEE");  
 setVar.insert(setVar.begin(), "FFFFFFF");  
  
 // 迭代器 循环  
 for (set<string>::iterator iteratorVar = setVar.begin(); iteratorVar != setVar.end(); iteratorVar++) {  
 cout << "循环item:" << \*iteratorVar << "\t";  
 // 循环item:AAAAAAA 循环item:BBBBBBB 循环item:CCCCCCC 循环item:DDDDDDD 循环item:EEEEEEE 循环item:FFFFFFF  
  
 // 循环item:FFFFFFF 循环item:EEEEEEE 循环item:DDDDDDD 循环item:CCCCCCC 循环item:BBBBBBB 循环item:AAAAAAA  
 }  
  
  
 return 0;  
}

// 专业的C++工程师开发有用 ，知识补充： 对象存入到容器后 对象的生命周期 状态  
  
#include <iostream>  
#include <set> // set 存入对象 奔溃（set会自动排序，对象没法排序，所以奔溃） 解决方案：自定义仿函数解决  
#include <vector> // 存入对象  
  
using namespace std;  
  
class Person {  
private:  
 string name;  
public:  
 Person(string name) : name(name) {}  
  
 void setName(string name) {  
 this->name = name;  
 }  
  
 string getName() {  
 return this->name;  
 }  
  
 Person(const Person &person) {  
 this->name = person.name; // 浅拷贝  
  
 cout << "Person拷贝构造函数执行了..." << endl;  
 }  
  
 ~Person() {  
 cout << "Person析构函数执行了" << endl;  
 }  
};  
  
int main() {  
 // Java：把对象存入 添加到 集合  
 // C++: 调用拷贝构造函数，存进去的是另一个新的对象  
  
 vector<Person> vectorVar;  
  
 // person 被main函数弹栈 析构一次  
 Person person("Derry"); // 2 Derry  
  
 // 里面的insert函数弹栈 析构一次  
 vectorVar.insert(vectorVar.begin(), person); // 外面的person是旧地址，到insert函数里面的person就是新地址（拷贝构造函数 一次）  
  
 person.setName("Kevin"); // 1  
  
 // newPerson 被main函数弹栈 析构一次  
 Person newPerson =  
 vectorVar.front(); // front里面的person是旧地址， 外面的newPerson就是新地址（拷贝构造函数 一次）  
  
 cout << "newPerson:" << newPerson.getName().c\_str() << endl;  
  
 // 3次析构函数 两次拷贝构造  
  
 return 0;  
} // main弹栈

// C++ 预定义函数（C++ 内置函数）  
  
#include <iostream>  
#include <set> // STL包  
#include <algorithm> // 算法包  
using namespace std;  
  
int main() {  
 // "Derry" + "AAAA" // 运算符重载  
  
 // C++已经提供了 预定义函数 plus,minus,multiplies,divides,modulus ...  
 plus<int> add\_func;  
  
 int r = add\_func(1, 1);  
 cout << r << endl;  
  
 plus<string> add\_func2;  
 string r2 = add\_func2("AAAA", "BBB");  
 cout << r2 << endl;  
  
 plus<float> add\_func3;  
 float r3 = add\_func3(4354.45f, 34.3f);  
 cout << r3 << endl;  
  
 return 0;  
}

// 手写预定义函数  
#include <iostream>  
#include <set> // STL包  
#include <algorithm> // 算法包  
using namespace std;  
  
template<typename Arg1, typename Arg2, typename Result>  
struct binary\_function  
{  
 /// 第一个参数类型 是底一个参数的类型  
 typedef Arg1 first\_argument\_type;  
  
 //econd\_argument\_type是第二个参数的类型  
 typedef Arg2 second\_argument\_type;  
  
 /// @c result\_type是返回类型  
 typedef Result result\_type;  
};  
  
// *TODO 对象 + 对象*// 1.运算符重载  
// 2.对象+对象 自己去写仿函数  
  
template<typename T>  
struct plus\_d /\*: public binary\_function<T, T, T>\*/  
{  
 T operator() (const T & x, const T & y) {  
 return x + y;  
 }  
};  
  
int main() {  
  
 plus\_d<int> add\_func;  
 int r = add\_func(1, 1);  
 cout << r << endl;  
  
 plus\_d<string> add\_func2;  
 string r2 = add\_func2("AAAA", "BBB");  
 cout << r2 << endl;  
  
 plus\_d<float> add\_func3;  
 float r3 = add\_func3(4354.45f, 34.3f);  
 cout << r3 << endl;  
  
 return 0;  
}

## C++高级之C++算法包源码全盘阅读与算法包实战

// 1.C++的函数适配器。 第一版本  
  
#include <iostream>  
#include <set> // stl包  
#include <algorithm> // 算法包  
using namespace std;  
  
int main() {  
 // std::cout << "算法包" << std::endl;  
  
 set<string, less<string>> setVar;  
 setVar.insert("AAAA");  
 setVar.insert("BBBB");  
 setVar.insert("CCCC");  
  
 for (set<string>::iterator iteratorVar = setVar.begin(); iteratorVar != setVar.end() ; iteratorVar++) {  
 cout << \*iteratorVar << endl;  
 }  
  
 // find\_if  
 // equal\_to 比较用的  
 set<string, less<string>>::iterator iteratorResult =  
  
 // 解决尴尬的问题 equal\_to 需要比较的 内容没有 使用 函数适配器 解决  
 // 现在的问题是： 没有办法把 CCCC 传递给 const \_Tp& \_\_y，就没法去比较  
 // find\_if(setVar.begin(), setVar.end(), equal\_to<string>("CCCC"), "CCCC");  
  
 // 使用函数适配器后，就能够 CCCC 传递给了 const \_Tp& \_\_y，  
 // setVar.begin(), setVar.end() 会把这些元素取出来 const \_Tp& \_\_x  
 // x == y 的比较  
 find\_if(setVar.begin(), setVar.end(), bind2nd(equal\_to<string>(), "CCCC"));  
  
 if (iteratorResult != setVar.end()) {  
 cout << "查找到了" << endl;  
 } else {  
 cout << "没有查找到" << endl;  
 }  
  
 return 0;  
}

// 1.C++的函数适配器。 第二版本  
  
#include <iostream>  
#include <set> // stl包  
#include <algorithm> // 算法包  
using namespace std;  
  
int main() {  
 // std::cout << "算法包" << std::endl;  
  
 set<string, less<string>> setVar;  
 setVar.insert("AAAA");  
 setVar.insert("BBBB");  
 setVar.insert("CCCC");  
  
 for (set<string>::iterator iteratorVar = setVar.begin(); iteratorVar != setVar.end() ; iteratorVar++) {  
 cout << \*iteratorVar << endl;  
 }  
  
 // bind2nd() 第二个参数 适配过去 （需要知道的）  
 // bind1st() 第一个参数 适配过去（NDK 很少用）  
  
 // 死活不想用 函数适配器 【Derry 没法做到，门都找不到】  
 find\_if(setVar.begin(), setVar.end(), equal\_to<string>());  
  
 return 0;  
}

// *TODO for\_each 算法包里面的遍历*// 算法包  
#include <iostream>  
#include <vector> // stl包  
#include <algorithm> // 算法包  
using namespace std;  
  
class \_\_F {  
public:  
 void operator() (int \_\_first) {  
 cout << "自定义一元谓词:" << \_\_first << endl;  
 }  
};  
  
int main() {  
 vector<int> vectorVar;  
 vectorVar.insert(vectorVar.begin(), 10000);  
 vectorVar.insert(vectorVar.begin(), 20000);  
 vectorVar.insert(vectorVar.begin(), 30000);  
 vectorVar.insert(vectorVar.begin(), 40000);  
 vectorVar.insert(vectorVar.begin(), 50000);  
 vectorVar.insert(vectorVar.begin(), 60000);  
  
 for\_each(vectorVar.begin(), vectorVar.end(), \_\_F());  
  
 return 0;  
}

// *TODO transform 算法包里面的 变化操作*#include <iostream>  
#include <vector> // stl包  
#include <algorithm> // 算法包  
using namespace std;  
  
class \_\_unary\_op {  
public:  
 int operator() (const int \_\_first) {  
 return \_\_first + 9; // 修改每个元素 +9  
 }  
};  
  
int main() {  
 vector<int> vectorVar;  
 vectorVar.insert(vectorVar.begin(), 10000);  
 vectorVar.insert(vectorVar.begin(), 20000);  
 vectorVar.insert(vectorVar.begin(), 30000);  
 vectorVar.insert(vectorVar.begin(), 40000);  
  
 // *TODO 第一种方式 类似于 RxJava map 变化操作符 【不看API，直接看算法包源码 印象非常深刻的】* // 迭代器 result == 参数三  
 transform(vectorVar.begin(), vectorVar.end(), vectorVar.begin(), \_\_unary\_op());  
  
 for (auto it = vectorVar.begin(); it != vectorVar.end() ; it++) {  
 cout << "第一种方式:" << \*it << endl;  
 }  
 cout << endl;  
  
 // 第三个参数接收返回值有啥用？没感受出来  
 // ==================================================================================  
  
 // *TODO 第二种方式* vector<int> vectorVarResult; // vectorVarResult 大小空间  
 vectorVarResult.resize(vectorVar.size());  
 transform(vectorVar.begin(), vectorVar.end(), vectorVarResult.begin(), \_\_unary\_op());  
  
 for (auto it = vectorVarResult.begin(); it != vectorVarResult.end() ; it++) {  
 cout << "第二种方式:" << \*it << endl;  
 }  
  
 return 0;  
}

// *TODO find find\_if 算法包里面的 查找元素*#include <iostream>  
#include <vector> // stl包  
#include <algorithm> // 算法包  
using namespace std;  
  
// find 没有自定义仿函数  
//int main() {  
// vector<int> vectorVar;  
// vectorVar.insert(vectorVar.begin(), 10000);  
// vectorVar.insert(vectorVar.begin(), 20000);  
// vectorVar.insert(vectorVar.begin(), 30000);  
// vectorVar.insert(vectorVar.begin(), 40000);  
//  
// // find 没有自定义仿函数  
// auto iteratorVar = find(vectorVar.begin(), vectorVar.end(), 40000);  
// if (iteratorVar != vectorVar.end()) {  
// cout << "查找到了" << endl;  
// } else {  
// cout << "没有找到" << endl;  
// }  
// return 0;  
//  
// /\*  
// ...  
// find(\_InputIterator \_\_first, 开始位置 迭代器  
// \_InputIterator \_\_last, 结束位置 迭代器  
// const \_Tp& \_\_val) 需要查找的元素 40000  
// {  
// ....  
// 结论：对\_\_find\_if的封装而已  
// return std::\_\_find\_if(\_\_first, \_\_last,  
// \_\_gnu\_cxx::\_\_ops::\_\_iter\_equals\_val(\_\_val));  
// }\*/  
//}  
  
  
// find\_if 有自定义仿函数  
class \_\_pred {  
public:  
 int number;  
 \_\_pred(int number) : number(number) {}  
 bool operator() (const int value) {  
 return number == value;  
 }  
};  
  
int main() {  
 vector<int> vectorVar;  
 vectorVar.insert(vectorVar.begin(), 10000);  
 vectorVar.insert(vectorVar.begin(), 20000);  
 vectorVar.insert(vectorVar.begin(), 30000);  
 vectorVar.insert(vectorVar.begin(), 40000);  
  
 auto it = find\_if(vectorVar.begin(), vectorVar.end(), \_\_pred(30000));  
  
 if (it != vectorVar.end()) {  
 cout << "查找到了" << endl;  
 } else {  
 cout << "没有找到" << endl;  
 }  
 return 0;  
  
 /\* 知道怎么阅读算法包源码 1  
 ...  
 find\_if(\_InputIterator \_\_first, 开始位置 迭代器  
 \_InputIterator \_\_last, 结束位置 迭代器  
 \_Predicate \_\_pred) 自定义仿函数  
 {  
 ... 监测工作而已  
 return std::\_\_find\_if(\_\_first, \_\_last,  
 \_\_gnu\_cxx::\_\_ops::\_\_pred\_iter(\_\_pred));  
 }  
  
 ...  
 \_\_find\_if(\_InputIterator \_\_first, 开始位置 迭代器  
 \_InputIterator \_\_last, 结束位置 迭代器  
 \_Predicate \_\_pred, *TODO 自定义仿函数*  
 ....)  
 {  
 while (\_\_first != \_\_last && !\_\_pred(\_\_first)) // \_\_pred(\_\_first) 自定义仿函数 怎么写 返回值bool 传入int类型  
 ++\_\_first; // 迭代器从开始位置挪动 算法思路： 指针++  
 return \_\_first;  
 }  
 \*/  
  
}

// *TODO count count\_if 算法包里面的 统计元素的个数*#include <iostream>  
#include <vector> // stl包  
#include <algorithm> // 算法包  
  
using namespace std;  
  
// count 没有自定义仿函数  
int main() {  
 vector<int> vectorVar;  
 vectorVar.push\_back(1);  
 vectorVar.push\_back(2);  
 vectorVar.push\_back(3);  
 vectorVar.push\_back(2);  
 vectorVar.push\_back(4);  
 vectorVar.push\_back(6);  
 vectorVar.push\_back(8);  
 vectorVar.push\_back(1);  
  
 int number = count(vectorVar.begin(), vectorVar.end(), 2);  
 cout << "等于2的个数是:" << number << endl;  
  
 // C++ 源码 函数适配器  
 number = count\_if(vectorVar.begin(), vectorVar.end(), bind2nd(less<int>(), 2)); // 函数适配器 配合 less <  
 cout << "等于2的个数是:" << number << endl;  
  
 number = count\_if(vectorVar.begin(), vectorVar.end(), bind2nd(greater<int>(), 2)); // 函数适配器 配合 less >  
 cout << "等于2的个数是:" << number << endl;  
  
 number = count\_if(vectorVar.begin(), vectorVar.end(), bind2nd(equal\_to<int>(), 2)); // 函数适配器 配合 less =  
 cout << "等于2的个数是:" << number << endl;  
  
 return 0;  
  
 // count\_if 源码分析...  
 /\*\*  
 ....  
 count\_if(\_InputIterator \_\_first, 迭代器 开始位置  
 \_InputIterator \_\_last, 迭代器 结束位置  
 \_Predicate \_\_pred) 自定义仿函数 \_\_pred在源码里面可以知道 我们去写自定义仿函数的规则  
 {  
 .... 省略 监测工作而已  
 return std::\_\_count\_if(\_\_first, \_\_last,  
 \_\_gnu\_cxx::\_\_ops::\_\_pred\_iter(\_\_pred));  
 }  
  
 ....  
 \_\_count\_if(\_InputIterator \_\_first,  
 \_InputIterator \_\_last,  
 \_Predicate \_\_pred)  
 {  
 typename iterator\_traits<\_InputIterator>::difference\_type \_\_n = 0; int \_\_n  
 for (; \_\_first != \_\_last; ++\_\_first) 思路：迭代器 ++ 挪动位置  
 if (\_\_pred(\_\_first)) 自定义仿函数 返回bool类型 ？？？ 迭代器类型  
 ++\_\_n;  
 return \_\_n; // 最终 count\_if 是返回int类型 \_\_n ++后的 统计元素的个数  
 }  
 \*/  
}

// *TODO merge 算法包里面的 对我们两个容器 合并成一个*#include <iostream>  
#include <vector> // stl包  
#include <algorithm> // 算法包  
  
using namespace std;  
  
int main() {  
 vector<int> vectorVar1;  
 vectorVar1.push\_back(10);  
 vectorVar1.push\_back(20);  
 vectorVar1.push\_back(30);  
 vectorVar1.push\_back(40);  
  
 vector<int> vectorVar2;  
 vectorVar2.push\_back(50);  
 vectorVar2.push\_back(60);  
 vectorVar2.push\_back(70);  
 vectorVar2.push\_back(80);  
  
 // 合并成一个容器 result  
 vector<int> vectorResult;  
 vectorResult.resize(vectorVar1.size() + vectorVar2.size());  
  
 merge(vectorVar1.begin(), vectorVar1.end(), vectorVar2.begin(), vectorVar2.end(), vectorResult.begin());  
 for (auto itVar = vectorResult.begin(); itVar != vectorResult.end() ; itVar++) {  
 cout << \*itVar <<endl;  
 }  
  
 return 0;  
  
 /\*\*  
  
 ...  
 merge(\_InputIterator1 \_\_first1, \_InputIterator1 \_\_last1, 第一个容器 位置  
 \_InputIterator2 \_\_first2, \_InputIterator2 \_\_last2, 第二个容器 位置  
 \_OutputIterator \_\_result) 最终合并后的结果  
 {  
 .... 监测工作而已  
 return \_GLIBCXX\_STD\_A::\_\_merge(\_\_first1, \_\_last1,  
 \_\_first2, \_\_last2, \_\_result,  
 \_\_gnu\_cxx::\_\_ops::\_\_iter\_less\_iter());  
 }  
  
 template<typename \_InputIterator1, typename \_InputIterator2,  
 typename \_OutputIterator, typename \_Compare>  
 \_GLIBCXX20\_CONSTEXPR  
 \_OutputIterator  
 \_\_merge(\_InputIterator1 \_\_first1, \_InputIterator1 \_\_last1, 第一个容器 位置  
 \_InputIterator2 \_\_first2, \_InputIterator2 \_\_last2, 第二个容器 位置  
 \_OutputIterator \_\_result, 最终合并后的结果  
 \_Compare \_\_comp)  
 {  
 while (\_\_first1 != \_\_last1 && \_\_first2 != \_\_last2)  
 {  
  
 做 合并 算法处理工作  
 if (\_\_comp(\_\_first2, \_\_first1))  
 {  
 \*\_\_result = \*\_\_first2;  
 ++\_\_first2;  
 }  
 else  
 {  
 \*\_\_result = \*\_\_first1;  
 ++\_\_first1;  
 }  
 ++\_\_result;  
 }  
  
 这个可以放弃 【不追逐了】  
 return std::copy(\_\_first2, \_\_last2,  
 std::copy(\_\_first1, \_\_last1, \_\_result));  
 }  
  
 \*/  
}

// *TODO sort 算法包里面的 对容器进行排序*#include <iostream>  
#include <vector> // stl包  
#include <algorithm> // 算法包  
  
using namespace std;  
  
int main() {  
 vector<int> vectorVar;  
 vectorVar.push\_back(10);  
 vectorVar.push\_back(30);  
 vectorVar.push\_back(20);  
  
 // if (\_\_comp(\_\_i, \_\_first)) 自定义仿函数规则 返回值 bool 第一个参数int 第二个参数 是int 吗  
  
 // 内置 的 仿函数 less<int>()  
 // less<int>() 里面泛型==函数模版 没法确定好 第二个参数的类型【到底是什么类型？】  
  
 // sort(vectorVar.begin(), vectorVar.end(), less<int>());  
 sort(vectorVar.begin(), vectorVar.end(), greater<int>());  
  
 // 直接打印 vectorVar容器 此时 是不是就已经排序了  
 for (auto itVar = vectorVar.begin(); itVar != vectorVar.end() ; itVar++) {  
 cout << \*itVar <<endl;  
 }  
}

// *TODO random\_shuffle 算法包里面的 随机打乱元素的顺序*#include <iostream>  
#include <vector> // stl包  
#include <algorithm> // 算法包  
  
using namespace std;  
  
int main() {  
 vector<int> vectorVar; // vector默认是没有排序功能的，默认输出： 65 53 84  
 vectorVar.push\_back(65);  
 vectorVar.push\_back(53);  
 vectorVar.push\_back(84);  
 vectorVar.push\_back(11);  
 vectorVar.push\_back(22);  
 vectorVar.push\_back(33);  
 vectorVar.push\_back(44);  
  
  
 sort(vectorVar.begin(), vectorVar.end(), less<int>()); // 排序后 53 65 82  
  
 random\_shuffle(vectorVar.begin(), vectorVar.end());  
  
 // 直接打印 vectorVar容器 此时 是不是就已经排序了  
 for (auto itVar = vectorVar.begin(); itVar != vectorVar.end() ; itVar++) {  
 cout << \*itVar << "\t";  
 // 84 65 53  
 }  
 return 0;  
}

// *TODO copy 算法包里面的 copy容器1的元素 到 容器2 （常用）*#include <iostream>  
#include <vector> // stl包  
#include <algorithm> // 算法包  
  
using namespace std;  
  
int main() {  
 vector<int> vectorVar; // vector默认是没有排序功能的，默认输出： 65 53 84  
 vectorVar.push\_back(100);  
 vectorVar.push\_back(200);  
 vectorVar.push\_back(300);  
 vectorVar.push\_back(400);  
 vectorVar.push\_back(500);  
 vectorVar.push\_back(600);  
 vectorVar.push\_back(700);  
  
 vector<int> vectorResult;  
 vectorResult.resize(vectorVar.size());  
  
 copy(vectorVar.begin(), vectorVar.end(), vectorResult.begin());  
 // 100 200 300 400 500 600 700  
  
 // 直接打印 vectorVar容器 此时 是不是就已经排序了  
 for (auto itVar = vectorResult.begin(); itVar != vectorResult.end() ; itVar++) {  
 cout << \*itVar << "\t";  
 }  
  
 // 源码变了 VS里面  
  
 return 0;  
}

// *TODO replace 算法包里面的 替换元素内容*#include <iostream>  
#include <vector> // stl包  
#include <algorithm> // 算法包  
  
using namespace std;  
  
int age;  
  
void set(int age) {  
 ::age = age;  
}  
  
int main() {  
 vector<int> vectorVar; // vector默认是没有排序功能的，默认输出： 65 53 84  
 vectorVar.push\_back(100);  
 vectorVar.push\_back(200);  
 vectorVar.push\_back(300);  
 vectorVar.push\_back(400);  
 vectorVar.push\_back(500);  
 vectorVar.push\_back(600);  
  
 // 100 ~ 200 范围  
 // replace(vectorVar.begin(), vectorVar.begin() + 2, 200, 222);  
  
 // 所有范围  
 replace(vectorVar.begin(), vectorVar.end(), 300, 333);  
  
 // 直接打印 vectorVar容器 此时 是不是就已经排序了  
 // Java有这个能力吗，想都不要想 指针  
 for (auto itVar = vectorVar.begin(); itVar != vectorVar.end() ; itVar++) {  
 cout << \*itVar << "\t";  
 }  
  
 return 0;  
}

## C++高级之引用进阶与多线程

1.引用进阶。

// *TODO 1.引用进阶。*#include <iostream>  
#include <vector>  
using namespace std;  
  
// *TODO =========================== [引用的本质就是指针]*/\*// *TODO 指针 互相换两个数*  
int changeNumber(int \*n1, int \*n2) {  
 int temp = \*n1;  
 \*n1 = \*n2;  
 \*n2 = temp;  
}  
  
// *TODO 引用 互相换两个数 [引用的本质就是指针]*  
int changeNumber2(int & n1, int & n2) { // 隐式：int \* n1, int \* n2  
 int temp = n1; // int temp = \*n1;  
 n1 = n2; // \*n1 = \*n2;  
 n2 = temp; // \*n2 = temp;  
}  
  
int main() {  
 int number1 = 100;  
 int number2 = 200;  
 // changeNumber(&number1, &number2);  
  
 // 我下面不需要去地址给他，为什么也可以，C、C++编译器 已经帮我们做了，所以无需写 &num 直接num  
 changeNumber2(number1, number2);  
  
 cout << number1 << " , " << number2 << endl;  
  
 return 0;  
}\*/  
  
// *TODO ============================ [左值 右值 引用]*class Student {  
private:  
 string info = "AAA"; // 旧变量  
  
 // *TODO 第一种情况【getInfo函数的info 与 main函数的result 是旧与新的两个变量而已，他们是值传递，所以右值修改时，影响不了里面的旧变量】*public:  
 string getInfo() {  
 return this->info;  
 }  
  
 // *TODO 第二种情况【getInfo函数的info 与 main函数的result 是引用关系，一块内存空间 有多个别名而已，所以右值修改时，直接影响旧变量】*public:  
 string & getInfo\_front() {  
 return this->info;  
 }  
};  
  
int main() {  
 /\*vector<int> v;  
 int r = v.front(); // 左值 获取  
 v.front() = 88; // 右值 修改\*/  
  
 Student student;  
  
 // *TODO 第一种情况* student.getInfo() = "九阳神功";  
 string result = student.getInfo();  
 cout << "第一种情况:" << result << endl;  
  
 // *TODO 第二种情况* student.getInfo\_front() = "三分归元气"; // 右值 修改内容  
 result = student.getInfo\_front(); // 左值 获取内容  
 cout << "第二种情况:" << result << endl;  
}

2.C++自带Thread线程。

// *TODO C++ 11 后出现的 自带 Thread*#include <iostream>  
#include <thread> // C++ 11 后出现的 自带 Thread 不是我们的重头戏，只是过一下  
#include <unistd.h>  
  
using namespace std;  
  
// 异步线程 子线程  
void runAction(int number) { // 相当于 Java的 run函数一样  
 for (int i = 0; i < 10; ++i) {  
 cout << "runAction：" << number << endl;  
 sleep(1);  
 }  
}  
  
// main函数的线程  
int main() {  
 // *TODO 方式一 main只等3秒钟，各种玩各种的，老死不相往来* /\*thread thread1(runAction, 100);  
  
 // sleep(3); // 我只等你三秒钟  
 cout << "main弹栈了" << endl;\*/  
  
 // *TODO 方式二 我等你执行完成后，我再执行* thread thread2(runAction, 100);  
 thread2.join(); // 我等runAction执行完成后，我再继续执行下面代码..  
 cout << "main弹栈了" << endl;  
 /\*  
 runAction：100  
 runAction：100  
 runAction：100  
 runAction：100  
 runAction：100  
 runAction：100  
 runAction：100  
 runAction：100  
 runAction：100  
 runAction：100  
 main弹栈了  
 \*/  
  
 return 0;  
}

3.Pthreads线程。

// *TODO pthreads 我们必须掌握的内容 重头戏*// *TODO pthreads 最简单的案例*#include <iostream>  
#include <pthread.h> // Derry Cygwin 有 pthreads支持  
using namespace std;  
  
// void \*(\*)(void \*)  
void \* customPthreadTask(void \* pVoid) { // 异步线程 相当于Java的Thread.run函数一样  
 // C++转换static\_cast 转换指针操作的  
 int \* number = static\_cast<int \*>(pVoid); // pVoid==number int的地址，所以我用int\*接收，很合理  
 cout << "异步线程执行了:" << \*number << endl;  
  
 return 0; // 坑 坑 坑，必须返回，否则有错误，不好查询  
}  
  
int main() {  
 int number = 9527;  
  
 /\*\*  
 int pthread\_create (pthread\_t \*, // 参数一：线程ID  
 const pthread\_attr\_t \*, // 参数二：线程属性  
 void \*(\*)(void \*), // 参数三：函数指针的规则  
 void \*); // 参数四：给函数指针传递的内容，void \* 可以传递任何内容  
 \*/  
 pthread\_t pthreadID; // 线程ID，每个线程都需要有的线程ID  
  
 pthread\_create(&pthreadID, 0, customPthreadTask, &number);  
  
 return 0;  
}

// *TODO pthread 的 三种情况分析 的*// *第一种情况，main函数只要结束，不等异步线程，全部技术*// *第二种情况，我们开发者，千万不要让 main函数睡眠的方式，去等待异步线程*// *第三种情况，main函数一直等待 异步线程，只有异步线程执行完成后，我在执行 join后面的代码*#include <iostream>  
#include <pthread.h> // Derry Cygwin 有 pthreads支持  
#include <unistd.h>  
  
using namespace std;  
  
// void \*(\*)(void \*)  
void \* runTask(void \* pVoid) { // 异步线程 子线程  
 int number = \*static\_cast<int \*>(pVoid);  
 cout << "异步线程执行了:" << number << endl;  
  
 for (int i = 0; i < 10; ++i) {  
 cout << "run:" << i << endl;  
 sleep(1);  
 }  
  
 return 0;  
}  
  
int main() {  
 int number = 999;  
  
 pthread\_t pthreadID;  
 pthread\_create(&pthreadID, 0, runTask, &number);  
  
 // 为了演示第二种情况  
 // sleep(3); // main函数只 异步线程三秒钟  
  
 // 异步线程在执行的过程中，我们的main函数 相当于 阻塞在这里不动了，只有异步线程执行完成后，我才开始执行join后面的代码  
 pthread\_join(pthreadID, 0);  
  
 cout << "main函数即将弹栈..." << endl;  
 return 0;  
}

4.Pthreads线程-分离线程与非分离线程概念。

// *TODO 分离线程 非分离线程 理论知识*// C++ 服务器开发，线程是非常要去高  
// C++ 分离线程 和 非分离线程 区别，应用场景？  
/\*\*  
 \* 答：分离线程： 各个线程都是自己运行自己的，老死不相往来，例如：main函数结束，全部结束，不会等待异步线程 【多线程情况下场景】  
 \* 非分离线程： 线程有协作的能力，例如：main函数线程会等待 异步线程执行完成后，我再执行 后面main函数的代码【协作，顺序执行 场景】  
 \*/  
  
#include <iostream>  
#include <pthread.h> // Derry Cygwin 有 pthreads支持  
#include <unistd.h>  
  
using namespace std;  
  
// void \*(\*)(void \*)  
void \* runTask(void \* pVoid) { // 异步线程 子线程  
 int number = \*static\_cast<int \*>(pVoid);  
 cout << "异步线程执行了:" << number << endl;  
  
 for (int i = 0; i < 10; ++i) {  
 cout << "run:" << i << endl;  
 sleep(1);  
 }  
 return 0;  
}  
  
int main() {  
 int number = 999;  
  
 pthread\_t pthreadID; // Cygwin允许有野指针  
 pthread\_create(&pthreadID, 0, runTask, &number);  
  
 pthread\_join(pthreadID, 0);  
  
 cout << "main函数即将弹栈..." << endl;  
 return 0;  
}

5.Pthreads多线程安全-互斥锁。

// *TODO C++ 互斥锁 == Java版本（synchronize） 多线程操作的安全 持有内置锁*#include <iostream>  
#include <pthread.h>  
#include <queue>  
#include <unistd.h> // sleep（秒）  
  
using namespace std;  
  
queue<int> queueData; // 定义一个全局的队列，用于 存储/获取  
  
pthread\_mutex\_t mutex; // 定义一个互斥锁，注意：（Cygwin平台 此互斥锁，不能有野指针，坑）  
  
// void \*(\*)(void \*)  
void \* task(void \* pVoid) {  
  
 /\*synchronize(锁) {  
 // code  
 }\*/  
  
 pthread\_mutex\_lock(&mutex); // 锁住  
  
 cout << "异步线程-当前线程的标记是:" << \*static\_cast<int \*>(pVoid) << "异步线程" << endl;  
  
 if (!queueData.empty()) { // 有元素  
 printf("异步线程-获取队列的数据:%d\n", queueData.front());  
 queueData.pop(); // 把数据弹出去，删除的意思  
 } else { // 没有元素  
 printf("异步线程-队列中没有数据了\n");  
 }  
  
 // sleep(0.2);  
  
 pthread\_mutex\_unlock(&mutex); // 解锁  
  
 return 0;  
}  
  
int main()  
{  
 // 初始化 互斥锁  
 pthread\_mutex\_init(&mutex, NULL);  
  
 // 给队列 初始化数据 手动增加数据进去  
 for (int i = 10001; i < 10011; ++i) {  
 queueData.push(i);  
 }  
  
 // 一次性定义10个线程  
 pthread\_t pthreadIDArray[10];  
 for (int i = 0; i < 10; ++i) {  
 pthread\_create(&pthreadIDArray[i], 0, task, &i);  
  
 // 不能使用 join，如果使用（就变成顺序的方式，就没有多线程的意义了，所以不能写join）  
 // pthread\_join(pthreadIDArray[i], 0);  
 }  
  
 // main函数等 异步线程  
 sleep(12);  
  
 // 销毁 互斥锁  
 pthread\_mutex\_destroy(&mutex);  
 cout << "main函数即将弹栈..." << endl;  
  
 // 每次运行 效果都不同：1，8，9，10，3，2，5，8  
 // 每次运行 效果都是错乱  
  
 return 0;  
}

// *TODO C++ 条件变量+互斥锁 == Java版本的（notify 与 wait 操作）*#pragma once  
  
#include <iostream>  
  
#include "safe\_queue\_too.h"  
using namespace std;  
SafeQueueClass<int> sq;  
  
// *TODO 模拟演示 消费者*void \* getMethod(void \*) {  
 while (true) {  
 printf("getMethod\n");  
  
 int value;  
 sq.get(value);  
 printf("消费者get 得到的数据:%d\n", value);  
  
 // 你只要传入 -1 就结束当前循环  
 if (-1 == value) {  
 printf("消费者get 全部执行完毕\n");  
 break;  
 }  
 }  
 return 0;  
}  
  
// *TODO 模拟演示 生产者*void \* setMethod(void \*) {  
 while (true) {  
 printf("setMethod\n");  
  
 int value;  
 printf("请输入你要生成的信息:\n");  
 cin >> value;  
  
 // 你只要传入 -1 就结束当前循环  
 if (-1 == value) {  
 sq.add(value); // 为了让消费者 可以结束循环  
 printf("消费者get 全部执行完毕\n");  
 break;  
 }  
  
 sq.add(value);  
 }  
 return 0;  
}  
  
int main() {  
 pthread\_t pthreadGet;  
 pthread\_create(&pthreadGet, 0, getMethod, 0);  
 // pthread\_join(pthreadGet, 0); 不能这样写，否则，下面的代码，可能无法有机会执行  
  
 pthread\_t pthreadSet;  
 pthread\_create(&pthreadSet, 0, setMethod, 0);  
  
  
 pthread\_join(pthreadGet, 0);  
  
 pthread\_join(pthreadSet, 0);  
  
  
 return 0;  
}

// *TODO #### 08-pthreads 手动依赖进来*#include <iostream> // windows平台 支持Windows  
using namespace std;  
#include <pthread.h> // minGW 默认是没有pthread的 *TODO 第四步 导入 pthread 的头文件*void \* run(void \* pVoid) {  
 for (int i = 0; i < 10; ++i) {  
 cout << "异步线程执行了一次:" << i << endl;  
 }  
 return 0;  
}  
  
int main() {  
  
 // JNI 去讲课 宏 预处理阶段的 先暂时这样理解：把宏理解常量  
  
 // *TODO 注意：必须是第五步完全没有任何问题了，才能用下面的代码，引出第六步* // *TODO 第六步 千辛万苦 很多坑 被我们折服了* pthread\_t pid; // 注意：pid可以允许存在野指针的行为，但是线程属性是绝对不允许的，线程属性必须初始化  
 pthread\_create(&pid, 0, run, 0);  
 pthread\_join(pid, 0);  
  
 cout << "main函数即将弹栈..." << endl;  
 return 0;  
}

## C++高级之手写智能指针与类型转换

1.智能指针的各种使用。

// *TODO 智能指针初探 内部机制原理*// 在真实开发过程中，才能体系智能指针的用途，否则写demo无法体现，为什么？  
// 因为真实开发过程中，很很多的代码，可能会导致你忘记写 delete 对象；  
// new 对象 我就马上 加入到 智能指针 我就不会忘记了  
  
#include <iostream>  
#include <memory> // 智能指针的头文件引入  
using namespace std;  
  
class Person {  
public:  
 ~Person() {  
 cout << "Person 析构函数" << endl;  
 }  
};  
  
int main() {  
 std::cout << "C++最后一节课" << std::endl;  
  
 Person \* person1 = new Person(); // 堆区开辟  
 Person \* person2 = new Person(); // 堆区开辟  
  
 // 以前：  
 // delete person1; 忘记写了，怎么办，非常严重的问题，没法释放  
  
 // 现在：  
 // shared\_ptr<Person> sharedPtr1(person1); // 智能指针帮你释放堆区开辟的 --> Person 析构函数  
  
 /\*  
 \*  
 \*  
 \*  
 \*  
 \*  
 \*  
 \*  
 \*  
 \*  
 \*  
 \* 10000行代码  
 \*  
 \*  
 \*  
 \*  
 \*  
 \*  
 \*  
 \*  
 \*  
 \*  
 \*  
 \*/  
 // 最后，就忘记给人家 delete person1  
 // 如果 加入到 智能指针，我就不用管了  
  
 shared\_ptr<Person> sharedPtr1(person1); // 栈区开辟sharedPtr1， 加1 等于1 引用计数  
 shared\_ptr<Person> sharedPtr2(person2); // 栈区开辟sharedPtr2 加1 等于1 引用计数  
  
 return 0;  
}  
// main函数弹栈，会释放 所有的栈成员 sharedPtr1 调用 sharedPtr1析构函数 减1 等于0 直接释放person1  
// main函数弹栈，会释放 所有的栈成员 sharedPtr2 调用 sharedPtr2析构函数 减1 等于0 直接释放person2

// *TODO 智能指针 使用频率高不高 1 2*// 智能指针 有循环依赖的问题，你要用就用好，不要用的复杂，循环依赖的问题  
// *TODO 智能指针 循环依赖问题，故意制作*#include <iostream>  
#include <memory> // 智能指针的头文件引入  
using namespace std;  
  
class Person2; // 先声明 Person2 让我们的Person1 直接找到  
  
class Person1 {  
public:  
 shared\_ptr<Person2> person2; // Person2智能指针 shared\_ptr 引用计数+1  
 ~Person1() {  
 cout << "Person1 析构函数" << endl;  
 }  
};  
  
class Person2 {  
public:  
 shared\_ptr<Person1> person1; // Person1智能指针 shared\_ptr 引用计数+1  
 ~Person2() {  
 cout << "Person2 析构函数" << endl;  
 }  
};  
  
  
int main() {  
 Person1 \* person1 = new Person1(); // 堆区开辟  
 Person2 \* person2 = new Person2(); // 堆区开辟  
  
 shared\_ptr<Person1> sharedPtr1(person1); // +1 = 1  
 shared\_ptr<Person2> sharedPtr2(person2); // +1 = 1  
  
 cout << "前 sharedPtr1的引用计数是:" << sharedPtr1.use\_count() << endl;  
 cout << "前 sharedPtr2的引用计数是:" << sharedPtr2.use\_count() << endl;  
  
 // 给Person2智能指针赋值  
 person1->person2 = sharedPtr2;  
 // 给Person1智能指针赋值  
 person2->person1 = sharedPtr1;  
  
 cout << "后 sharedPtr1的引用计数是:" << sharedPtr1.use\_count() << endl;  
 cout << "后 sharedPtr2的引用计数是:" << sharedPtr2.use\_count() << endl;  
  
 return 0;  
} // 减1 = 0 释放对象

// *TODO 智能指针 解决循环依赖的问题 weak 智能指针 没有引用计数*// C++ 11后 推出智能指针，为什么要推出？ JVM非常厉害，完全不用管对象的回收的问题  
  
#include <iostream>  
#include <memory> // 智能指针的头文件引入  
using namespace std;  
  
class Person2; // 先声明 Person2 让我们的Person1 直接找到  
  
class Person1 {  
public:  
 weak\_ptr<Person2> person2; // Person2智能指针 没有引用计数 无法+1  
 ~Person1() {  
 cout << "Person1 析构函数" << endl;  
 }  
};  
  
class Person2 {  
public:  
 weak\_ptr<Person1> person1; // Person1智能指针 没有引用计数 无法+1  
 ~Person2() {  
 cout << "Person2 析构函数" << endl;  
 }  
};  
  
  
int main() {  
 Person1 \* person1 = new Person1(); // 堆区开辟  
 Person2 \* person2 = new Person2(); // 堆区开辟  
  
 shared\_ptr<Person1> sharedPtr1(person1); // +1 = 1  
 shared\_ptr<Person2> sharedPtr2(person2); // +1 = 1  
  
 cout << "前 sharedPtr1的引用计数是:" << sharedPtr1.use\_count() << endl;  
 cout << "前 sharedPtr2的引用计数是:" << sharedPtr2.use\_count() << endl;  
  
 // 给Person2智能指针赋值  
 person1->person2 = sharedPtr2;  
 // 给Person1智能指针赋值  
 person2->person1 = sharedPtr1;  
  
 cout << "后 sharedPtr1的引用计数是:" << sharedPtr1.use\_count() << endl;  
 cout << "后 sharedPtr2的引用计数是:" << sharedPtr2.use\_count() << endl;  
  
 return 0;  
} // 减1 = 0 释放对象

// *TODO unique 智能指针 设计的够简单，每一那么多功能 [独占式智能指针]*#include <iostream>  
#include <memory> // 智能指针的头文件引入  
  
class Person {};  
  
int main() {  
 Person \* person1 = new Person();  
 Person \* person2 = new Person();  
  
 std::unique\_ptr<Person> uniquePtr1(person1);  
  
 // 严格禁止  
 // std::unique\_ptr<Person> uniquePtr2 = uniquePtr1; unique不允许，因为是独占的  
  
 // shared\_ptr 是可以的，会造成隐患问题  
  
 return 0;  
}

2.手写智能指针。

// *TODO C++ 期末考试 -- 手写智能指针*#include <iostream>  
#include <memory> // 智能指针的头文件引入  
  
using namespace std;  
  
class Person {};  
  
int main() {  
 Person \* person1 = new Person();  
 // Person \* person2 = new Person();  
  
 // shared\_ptr<Person> sharedPtr0;  
  
 shared\_ptr<Person> sharedPtr1(person1); // +1 引用计数  
  
 // 第一种情况 会调用拷贝构造函数  
 // shard\_ptr智能指针的特性  
 // shared\_ptr<Person> sharedPtr2 = sharedPtr1; // +1 再引用计数 不会调用构造函数，只能执行拷贝构造函数  
  
 // 第二种情况 不会调用拷贝构造函数  
 shared\_ptr<Person> sharedPtr2; // +1 再引用计数 先构造函数 开辟sharedPtr2对象  
  
 // 给sharedPtr2对象 从新赋值  
 sharedPtr2 = sharedPtr1; // 自定义 =号运算符重载  
  
  
 return 0;  
}  
// main函数弹栈 sharedPtr1栈成员 ---> 析构函数 --等于0 就释放对象

// *TODO C++ 期末考试 -- 手写智能指针 2*#include "CustomPtr.h"  
  
class Student {  
public:  
 ~Student() {  
 cout << "析构函数 释放Student" << endl;  
 }  
};  
  
// *TODO 智能指针内置的*void action() {  
 Student \*student1 = new Student();  
 Student \*student2 = new Student();  
  
 // *TODO 第一种情况* /\*shared\_ptr<Student> sharedPtr1(student1);  
 shared\_ptr<Student> sharedPtr2(student2);\*/  
  
 // *TODO 第二种情况* /\*shared\_ptr<Student> sharedPtr1 (student1);  
 shared\_ptr<Student> sharedPtr2 = sharedPtr1;\*/  
  
 // *TODO 通用的打印* /\*cout << "智能指针内置的 sharedPtr1:" << sharedPtr1.use\_count() << endl;  
 cout << "智能指针内置的 sharedPtr2:" << sharedPtr2.use\_count() << endl;\*/  
}  
  
// *TODO 手写的智能指针*void action2() {  
 Student \*student1 = new Student();  
 Student \*student2 = new Student();  
  
 // *TODO 第一种情况* /\*Ptr<Student> sharedPtr1(student1);  
 Ptr<Student> sharedPtr2(student2);\*/  
  
 // *TODO 第二种情况* /\*Ptr<Student> sharedPtr1 (student1);  
 Ptr<Student> sharedPtr2 = sharedPtr1;\*/  
  
 // *TODO 第二种情况* // *TODO 情况一* /\*Ptr<Student> sharedPtr1 (student1); // sharedPtr1构建对象  
 Ptr<Student> sharedPtr2; // sharedPtr2也会构建对象， 此对象指向了object 与 count，必须释放  
  
 // 在你写下面这个之前，我必须是发 sharedPtr2 的所有 object count 全部释放  
 // sharedPtr2完全释放干净后，我才放心然你赋值 sharedPtr2 = sharedPtr1  
 sharedPtr2 = sharedPtr1;\*/  
  
  
 // *TODO 情况二* Ptr<Student> sharedPtr1 (student1); // sharedPtr1构建对象  
 // student2 成为野对象（每一被智能指针管理的对象 称为 野对象）  
 Ptr<Student> sharedPtr2 (student2);  
  
 // 在你写下面这个之前，我必须是发 sharedPtr2 的 的 student2 全部释放成功  
 // sharedPtr2完全释放干净后,才放心然你赋值 sharedPtr2 = sharedPtr1  
  
 sharedPtr2 = sharedPtr1;  
  
 // delete student2; // 如果--哪些逻辑不写，就必须手动是否 student2  
  
 // *TODO 通用的打印* cout << "手写的智能指针 sharedPtr1:" << sharedPtr1.use\_count() << endl;  
 cout << "手写的智能指针 sharedPtr2:" << sharedPtr2.use\_count() << endl;  
}  
  
  
int main() {  
 cout << "下面是 C++内置智能指针 ===========" << endl;  
 // action();  
 cout << endl;  
  
 cout << "下面是 自定义的智能指针 ===========" << endl;  
 action2();  
}

3.四种类型转换。

// 3.四种类型转换。 const\_cast const修饰的 都可以去转换  
  
#include <iostream>  
  
using namespace std;  
  
class Person {  
public:  
 string name = "default";  
};  
  
int main() {  
 const Person \* p1 = new Person();  
 // p1->name = "Derry"; // 报错：常量指针，不写修改值  
  
 Person \* p2 = const\_cast<Person \*>(p1); // 转成 非常量指针  
 p2->name = "Derry";  
  
 cout << p1->name << endl;  
  
 return 0;  
}

// 3.四种类型转换。 static\_cast 指针相关的操作 可以用 static\_cast  
  
#include <iostream>  
using namespace std;  
  
class FuClass {  
public:  
 void show() {  
 cout << "fu show" << endl;  
 }  
};  
  
class ZiClass : public FuClass {  
public:  
 void show() {  
 cout << "zi show" << endl;  
 }  
};  
  
int main() {  
 int n = 88;  
 void \* pVoid = &n;  
 int \* number = static\_cast<int \*>(pVoid);  
 cout << \*number << endl;  
  
 // ====================  
  
 FuClass \* fuClass = new FuClass;  
 // fuClass->show();  
  
 // static\_cast(编译期) 看左边 ZiClass \*  
 ZiClass \* ziClass = static\_cast<ZiClass \*>(fuClass);  
 ziClass->show();  
  
 delete fuClass; // 回收规则：一定是谁new了，我就delete谁  
  
 return 0;  
}

// 3.四种类型转换。 dynamic 字符类多态 运行期 转换  
  
#include <iostream>  
using namespace std;  
  
class FuClass {  
public:  
 // 动态转换必须让父类成为虚函数  
 virtual void show() {  
 cout << "fu show" << endl;  
 }  
};  
  
class ZiClass : public FuClass {  
public:  
 void show() {  
 cout << "zi show" << endl;  
 }  
};  
  
int main() {  
 // 动态类型转换的时候，在运行期 由于fuClass 是new 父类的，已成定局，就不能转换子类  
 // FuClass \* fuClass = new FuClass(); // 失败  
  
 FuClass \* fuClass = new ZiClass; // 已成定局 是子类  
 ZiClass \* ziClass = dynamic\_cast<ZiClass \*>(fuClass);  
  
 // *TODO 子类转父类不行的，同学们自己去试一试* // 动态转换是有返回值， null 转换失败  
 if (ziClass) { // ziClass != null  
 cout << "恭喜，转换成功 " ;  
 ziClass->show();  
 } else {  
 cout << "不恭喜 转换失败" << endl ;  
 }  
  
  
  
 return 0;  
}

// 3.四种类型转换。 reinterpret\_cast 强制转换 比 static\_cast要强大， static\_cast能够做的事情，  
// reinterpret\_cast强制转换都可以，同时并且附加 新功能  
  
#pragma once  
#include <iostream>  
#include <iostream>  
#include <iostream>  
#include <iostream>  
#include <iostream>  
#include <iostream>  
  
using namespace std;  
  
class DerryPlayer {  
public:  
 void show() {  
 cout << "DerryPlayer" << endl;  
 }  
};  
  
int main() {  
 DerryPlayer \* derryPlayer = new DerryPlayer();  
 long playerValue = reinterpret\_cast<long>(derryPlayer); // 把对象变成数值  
  
 // 通过数值 变成对象  
 DerryPlayer \* derryPlayer2 = reinterpret\_cast<DerryPlayer \*>(playerValue);  
 derryPlayer2->show();  
  
 printf("derryPlayer:%p\n", derryPlayer);  
 printf("derryPlayer2:%p\n", derryPlayer2);  
  
 // 前面的只是：为什么不一样：因为指针存放地址，同时指针有自己的地址，而你打印了自己的的地址，能一样？  
 printf("derryPlayer:%p\n", &derryPlayer);  
 printf("derryPlayer2:%p\n", &derryPlayer2);  
  
}

4.nullptr。

#include <iostream>  
using namespace std;  
  
void show(int \* i) {  
 cout << " show(int \* i) " << endl;  
}  
  
void show(int i) {  
 cout << " show(int i) " << endl;  
}  
  
int main() {  
 show(9);  
  
 show(nullptr); // C++11 后的特性： 原本本意代替NULL，除了代替NULL，还有此功能  
  
 return 0;  
}

## JNI技术之JNI详讲交互与预处理器

// *TODO 预处理器不是编译器，预处理器主要完成文本替换的操作（文本替换，以后专门在Linux中去讲），预处理器都是用 #xxx 的写法，并不是注释哦*/\*  
 #include 导入头文件  
 #if if判断操作 【if的范畴 必须endif】  
 #elif else if  
 #else else  
 #endif 结束if  
 #define 定义一个宏  
 #ifdef 如果定义了这个宏 【if的范畴 必须endif】  
 #ifndef 如果没有定义这个宏 【if的范畴 必须endif】  
 #undef 取消宏定义  
 #pragma 设定编译器的状态  
 \*/  
  
#include <iostream>  
using namespace std;  
  
int main() {  
 // std::cout << "宏" << std::endl;  
  
#if 1 // if  
 cout << "真" << endl;  
  
#elif 0 // else if  
 cout << "假" << endl;  
  
#else  
 cout << "都不满足" << endl;  
  
#endif // 结束if  
 cout << "结束if" << endl;  
  
 return 0;  
}

#ifndef CLIONCPPPROJECT\_T2\_H // 如果没有定义这个宏 解决循环拷贝的问题  
#define CLIONCPPPROJECT\_T2\_H // 我就定义这个宏  
  
// 100 行代码  
// 第一次能够进来  
// 第二次 第n此进不来 直接 解决循环拷贝的问题了  
  
// ---------------  
#ifndef isRelease // 如果没有isRelease这个宏  
#define isRelease 1 // 是否是正式环境下 【我就定义isRelease这个宏】  
  
#if isRelease == true  
#define RELEASE // 正式环境下 定义RELEASE宏  
  
#elif isRelease == false  
#define DEBUG // 测试环境下 定义DEBUG宏  
  
#endif // 结束里面的if  
#endif // 结束里面的if  
  
#endif //CLIONCPPPROJECT\_T2\_H // 结束外面的if

#include <iostream>  
#include "T2.h"  
using namespace std;  
  
int main() {  
  
 // if 条件判断  
 // ifdef xxx 是否定义了xxx这个宏  
  
#ifdef DEBUG // 是否定义了DEBUG这个宏  
 cout << "在测试环境下，迭代功能" << endl;  
 // 省略 500行 ...  
  
#else RELEASE  
 cout << "在正式环境下，功能上下中" << endl;  
 // 省略 500行 ...  
  
#endif // 结束IF  
  
}

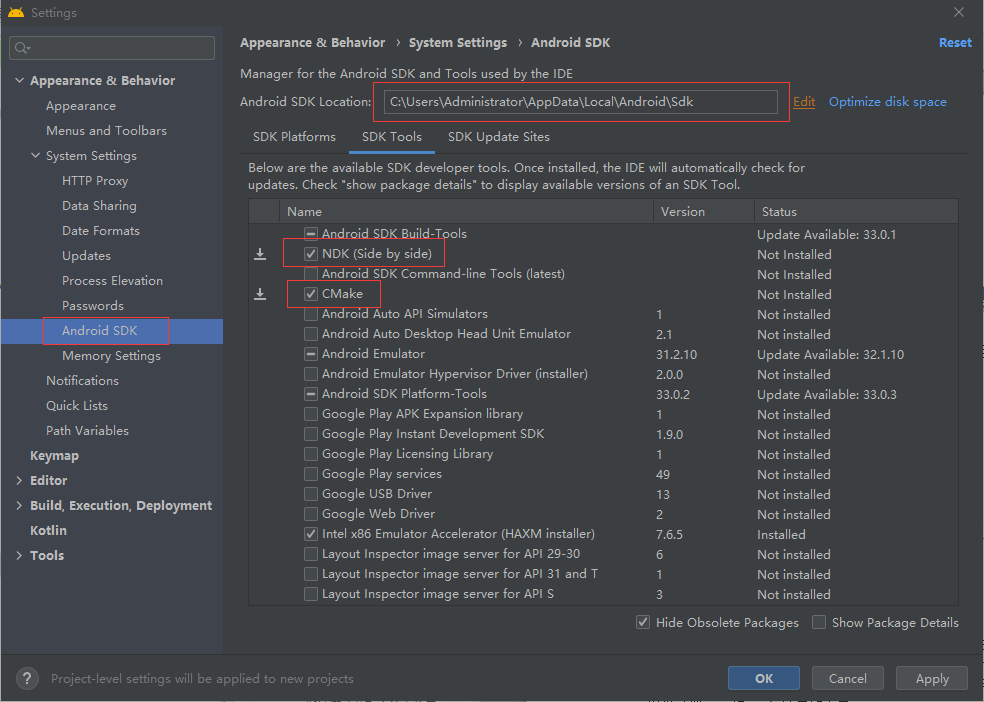
// 宏的取消 #undef 宏  
  
#include <iostream>  
using namespace std;  
  
int main() {  
 int i = 1  
  
#ifndef DERRY // 如果没有定义这个宏  
#define DERRY // 我就定义宏  
#ifdef DERRY // 是否定义了这个宏  
 for (int i = 0; i < 6; ++i) {  
 cout << "Derry 1" << endl;  
 }  
 // 省略 500行 ...  
  
#ifdef DERRY // 是否定义了这个宏  
 for (int i = 0; i < 6; ++i) {  
 cout << "Derry 2" << endl;  
 }  
 // 省略 500行 ...  
  
#undef DERRY // 取消宏的定义，下面的代码，就没法用这个宏了，相当于：没有定义过DERRY宏  
  
#ifdef DERRY  
 cout << "你定义了Derry宏" << endl;  
#else  
 cout << "你没有定义了Derry宏" << endl;  
  
#endif  
#endif  
#endif  
#endif  
  
 return 0;  
}

// 宏变量 真实开发中：宏都是大写  
  
#include <iostream>  
using namespace std;  
  
#define VALUE\_I 9527  
#define VALUE\_S "AAA"  
#define VALUE\_F 545.3f  
  
int main() {  
 int i = VALUE\_I; // 预处理阶段 宏会直接完成文本替换工作，替换后的样子：int i = 9527;  
 string s = VALUE\_S; // 预处理阶段 宏会直接完成文本替换工作，替换后的样子：string s = "AAA";  
 float f = VALUE\_F; // 预处理阶段 宏会直接完成文本替换工作，替换后的样子：float f = 545.3f;  
  
 return 0;  
}

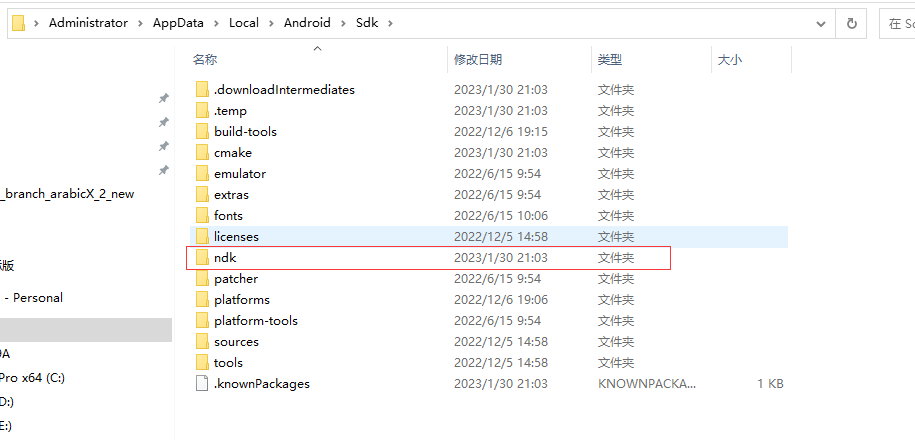
// 宏函数 真实开发中：宏函数都是大写  
#include <iostream>  
using namespace std;  
  
#define SHOW(V) cout << V << endl; // 参数列表 无需类型 返回值 看不到  
#define ADD(n1, n2) n1 + n2  
#define CHE(n1, n2) n1 \* n2 // 故意制作问题 ，宏函数的注意事项  
  
// 复杂的宏函数  
#define LOGIN(V) if(V==1) { \  
 cout << "满足 你个货输入的是:" << V << endl; \  
} else { \  
 cout << "不满足 你个货输入的是:" << V << endl; \  
} // 这个是结尾，不需要加 \  
  
void show() {}  
  
int main() {  
 SHOW(8);  
 SHOW(8.8f);  
 SHOW(8.99);  
  
 int r = ADD(1, 2);  
 cout << r << endl;  
  
 r = ADD(1+1, 2+2);  
 cout << r << endl;  
  
 // r = CHE(1+1, 2+2);  
 r = 1+1 \* 2+2; // 文本替换：1+1 \* 2+2 先算乘法 最终等于 5  
 cout << r << endl; // 我们认为的是8， 但是打印5  
  
 LOGIN(0);  
 LOGIN(0);  
 LOGIN(0);  
 LOGIN(0);  
 LOGIN(0);  
 LOGIN(0);  
 // 会导致代码体积增大  
  
 show();  
 show();  
 show();  
 show();  
 show();  
 // 普通函数，每次都会进栈 弹栈 ，不会导致代码体积增大  
  
 return 0;  
}  
  
// 宏函数  
/\*  
 \* 优点：  
 \* 1.文本替换，不会造成函数的调用开销(开辟栈空间，形参压栈，函数弹栈释放 ..)  
 \*  
 \* 缺点：  
 \* 1.会导致代码体积增大  
 \*  
 \*/

# Android studio 安装C++编译环境

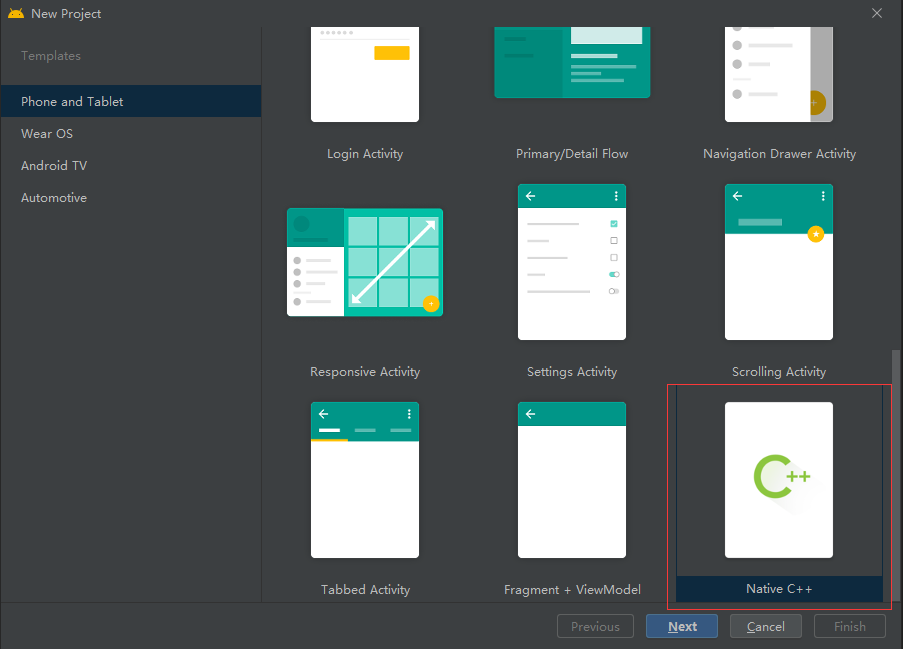
1. 新安装一个android studio版本，使用的是这个版本D:\android studio\android-studio\_2021.1.1.9。进入到android studio，找到SDK Manager点击进入，修改sdk的目录，不要和原来项目位置相同



下载ndk成功之后会存在下面的目录



创建so工程，新建工程选择Native C++



创建完成可以直接运行项目，运行之后可以直接获取C++的资源进行显示。

# JNI

## JNI

### JNI概述：

定义：Java Native Interface，即Java本地接口

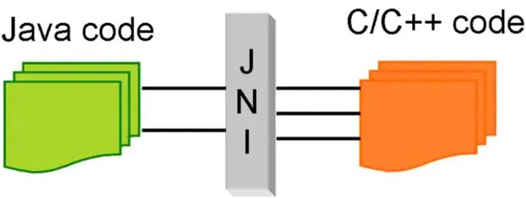
作用：使得Java 与 本地其他类型语言（如C、C++）交互

JNI是Java调用Native语言的一种特性JNI是属于Java的，与Android无直接关系。

实际中的驱动都是C/C++开发的,通过JNI,Java可以调用C/C++实现的驱动，从而扩展Java虚拟机的能力。另外在高效率的数学运算、游戏的实时渲染、音视频的编码和解码等方面，一般都是用C开发的。

### 为什么要有JNI

实际使用中，Java需要与本地代码进行交互，因为 Java 具备跨平台的特点，所以Java 与 本地代码交互的能力非常弱，采用JNI特性增强 Java 与本地代码交互的能力。



### JNI和NDK区别

#### JNI

JNI的全称是Java Native Interface，即本地Java接口。因为 Java 具备跨平台的特点，所以Java 与 本地代码交互的能力非常弱。采用JNI特性可以增强 Java 与本地代码交互的能力，使Java和其他类型的语言如C++/C能够互相调用。

注意：

JNI是 Java 调用 Native 语言的一种特性，Java与C/C++交互

JNI 是属于 Java 的，与 Android 无直接关系，Android程序使用JNI比Java程序使用更简单

#### NDK

Native Development Kit， Android的一个工具开发包。NDK集成了交叉编译器(交叉编译器需要UNIX或LINUX系统环境)，并提供了相应的mk文件隔离CPU、平台、ABI等差异，开发人员只需要简单修改mk文件。

NDK能帮助开发者快速开发C、 C++的动态库，将动态库编译成.so文件供Java调用，并支持将.so和应用一起打包成 apk。

为什么要使用NDK？

对于要求高性能的需求功能如游戏，采用C/C++运行的更加有效率

Java是半解释型语言，容易被反编译后得到源代码，而本地代码库.so不容易被反编译，安全程度高

功能扩展性好，可以使用除了java的其他开发语言的开源库，不依赖于Dalvik Java虚拟机的设计

便于平台移植，C/C++开发的代码可以移植到其他平台上使用

注意：

一般应用用不上NDK，它会增大开发过程的复杂性。

NDK是属于 Android 的，与Java并无直接关系

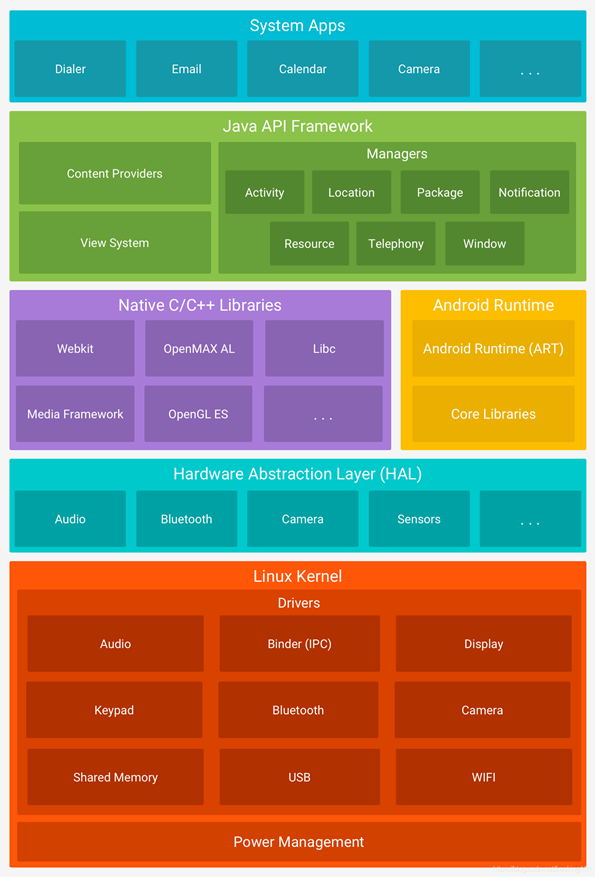
总结

NDK帮助将C/C++库打包成动态库.so，JNI是C/C++动态库能被调用的桥梁接口

JNI是java接口，用于Java与C/C++之间的交互，作为两者的桥梁

NDK是Android工具开发包，帮助快速开发C/C++动态库，相当于JDK开发java程序一样，同时能帮打包生成.so库

### Android 架构图



### 签名规则

java类型 属性类型符号

boolean Z

byte B

char C

short S

int I

long J

float F

double D

void V

object L完整的类名；

array[数组的数据类型 ---------- int[] [I double[][] [[D

method(参数类型)返回值类型 ---------- void name（int a，double b） （ID）V

### javap命令获取签名

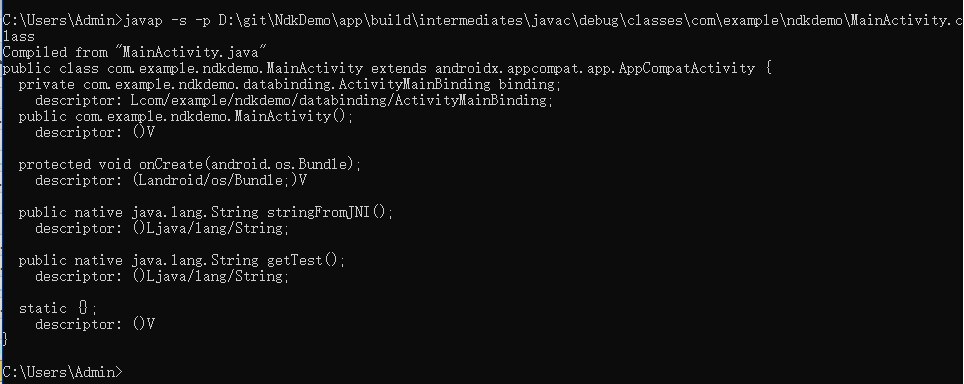
首先打开build\intermediates\javac\debug\classes\com\example\ndkdemo目录，找到需要查看签名的文件，这里选择MainActivity。



输入以下命令：

javap -s -p   
D:\git\NdkDemo\app\build\intermediates\javac\debug\classes\com\example\ndkdemo\MainActivity.class

就可以查看MainActivity下各个方法的签名：



### JNI函数细节介绍

JNI函数如下：

extern "C" JNIEXPORT jstring JNICALL  
Java\_com\_example\_ndkdemo\_MainActivity\_stringFromJNI(  
 JNIEnv\* env,  
 jobject /\* this \*/) {  
 std::string hello = "Hello from C++";  
 return env->NewStringUTF(hello.c\_str());  
}

extern “C”:表示下面的代码采用C的编译方式，因为JNIEve使用的是JNINativeInterface结构体，这个结构体里面的大概300个方法都是用的是C语言编写。

JNIEXPORT: JNI重要标记关键字，不能少,这里固定写成这样。

void: 表示Java中的void

JNICALL： 关键字，约束函数入栈顺序，和堆栈内存清理的规则

Java\_com\_example\_ndkdemo\_MainActivity\_stringFromJNI：标记本地方法名称

Java方法规则：

Java\_包名\_类名\_方法名

如果包名含有下划线，则生成的java方法使用1来表示原有的下划线。

例如包名是这个：com.derry.as\_jni\_project，则生成的本地方法名称是：

Java\_com\_derry\_as\_1jni\_1project\_ MainActivity\_stringFromJNI

JNIEnv \*env: 这个是 c 或C++和 java 相互调用的桥梁，他们之间调用必须通过JNIEnv。

jobject: java传递下来的对象

jclass: java传递下来的 class 对象

如果当前是C语言文件，JNIEnv通过下面的方式引用：

(\*env)->xxx函数

因为C语言的JNIEnv \*env是二级指针

如果当前是C++文件，JNIEnv通过下面的方式引用：

env->xxx函数

因为C++语言的 JNIEnv \*env 是一级指针。

## C++ 与 Java 交互操作

### 从C++获取String字符串

extern "C"  
JNIEXPORT   
jstring JNICALL  
Java\_com\_example\_ndkdemo\_BaseActivity\_stringFromJNI(JNIEnv \*env, jobject thiz) {  
 std::string hello = "Hello from C++";  
 return env->NewStringUTF(hello.c\_str());  
}

在Java代码调用：

String str = stringFromJNI();  
Utils.*log*(str);

上面的代码能从C++层获取字符串，并输出到控制台。

### 通过C++层修改Java本地属性

涉及api:

*/\*\*  
 \* 通过传入java完整类名，获取jclass对象  
 \** ***@param*** *java完整类名  
 \** ***@return*** *jclass  
 \*/*jclass FindClass(const char\* name)

*/\*\*  
 \* 获取变量的jfieldID对象  
 \** ***@param*** *clazz 具体类的jclass对象  
 \** ***@param*** *name 方法名  
 \** ***@param*** *sig 方法签名  
 \** ***@return*** *jfieldID对象  
 \*/*jfieldID GetFieldID(jclass clazz, const char\* name, const char\* sig)

*/\*\*  
 \* 将一个字符串转为jstring对象  
 \** ***@param*** *字符串  
 \** ***@return*** *jstring对象  
 \*/*jstring NewStringUTF(const char\* bytes)

*/\*\*  
 \* 设置Object类型变量的值  
 \** ***@param*** *obj 变量所属java类的jobject对象  
 \** ***@param*** *fieldID 变量fieldId  
 \** ***@param*** *value 设置值  
 \*/*void SetObjectField(jobject obj, jfieldID fieldID, jobject value)

测试代码如下：

/\*\*  
 \* 改变我们的属性name  
 \*/  
extern "C"  
JNIEXPORT void JNICALL  
Java\_com\_example\_ndkdemo\_BaseActivity\_changeName(JNIEnv \*env, jobject thiz) {  
 jclass mainActivityCls = env->FindClass("com/example/ndkdemo/BaseActivity"); // 方式一  
  
 // jfieldID GetFieldID(jclass clazz, const char\* name, const char\* sig)  
 jfieldID nameFid = env->GetFieldID(mainActivityCls, "name", "Ljava/lang/String;");  
  
 // void SetObjectField(jobject obj, jfieldID fieldID, jobject value)  
 jstring value = env->NewStringUTF("修改为Beyond");  
 env->SetObjectField(thiz, nameFid, value);  
}

### 修改Java静态变量

修改静态方法有一套专门为静态变量实现的api，静态方法不能使用非静态的api去设置。

涉及api:

*/\*\*  
 \* 静态方法获取变量id  
 \** ***@param*** *clazz 具体类的jclass对象  
 \** ***@param*** *name 方法名  
 \** ***@param*** *sig 方法签名  
 \** ***@return*** *jfieldID类型的方法id  
 \*/*jfieldID GetStaticFieldID(jclass clazz, const char\* name, const char\* sig)  
  
*/\*\*  
 \* 获取具体类型的变量  
 \** ***@param*** *clazz 具体类的jclass对象  
 \** ***@param*** *fieldID 方法id  
 \** ***@return*** *jint类型的方法变量  
 \*/*jint GetStaticIntField(jclass clazz, jfieldID fieldID)  
   
*/\*\*  
 \* 设置静态变量的值  
 \** ***@param*** *clazz 具体类的jclass对象  
 \** ***@param*** *fieldID 方法id  
 \** ***@param*** *value 设置值  
 \** ***@return*** *\*/*void SetStaticIntField(jclass clazz, jfieldID fieldID, jint value)

测试代码如下：

extern "C"  
JNIEXPORT void JNICALL  
Java\_com\_example\_ndkdemo\_BaseActivity\_changeAge(JNIEnv \*env, jclass mainActivityCls) {  
 // jfieldID GetStaticFieldID(jclass clazz, const char\* name, const char\* sig)  
 jfieldID ageFid = env->GetStaticFieldID(mainActivityCls, "age", "I");  
  
 // jint 背后就是int，所以可以直接用， 但是String 必须用 jstring  
 int age = env->GetStaticIntField(mainActivityCls, ageFid); // 获取之前的age  
  
 // void SetStaticIntField(jclass clazz, jfieldID fieldID, jint value)  
 env->SetStaticIntField(mainActivityCls, ageFid, age + 10);  
}

### 修改Java私有变量

设置java私有变量和普通变量的流程是一样的，下面通过另外一种方式获取jclass：  
通过GetObjectClass()方法获取jclass对象。

涉及api:

*/\*\*  
 \* 获取jclass对象  
 \** ***@param*** *obj object对象  
 \** ***@return*** *返回传入object示例的jclass对象  
 \*/*jclass GetObjectClass(jobject obj)  
  
*/\*\*  
 \* 设置double类型的属性  
 \** ***@param*** *obj 变量所属java类的jobject对象  
 \** ***@param*** *fieldID 变量fieldId  
 \** ***@param*** *value 设置值  
 \*/*void SetDoubleField(jobject obj, jfieldID fieldID, jdouble value)

测试代码：

extern "C"  
JNIEXPORT void JNICALL  
Java\_com\_example\_ndkdemo\_BaseActivity\_changeNumber(JNIEnv \*env, jobject mainActivitThis) {  
 jclass mainActivityCls = env->GetObjectClass(mainActivitThis); // 方式二  
 // jfieldID GetFieldID(jclass clazz, const char\* name, const char\* sig)  
 jfieldID numberFid = env->GetFieldID(mainActivityCls, "number", "D");  
 // void SetDoubleField(jobject obj, jfieldID fieldID, jdouble value)  
 env->SetDoubleField(mainActivitThis, numberFid, 885.55);  
  
 // 修改 final number2 ========================  
 jfieldID numberFid2 = env->GetFieldID(mainActivityCls, "number2", "D");  
 env->SetDoubleField(mainActivitThis, numberFid2, 7777.99);  
  
 double result = env->GetDoubleField(mainActivitThis, numberFid2);  
 // \_\_android\_log\_print(ANDROID\_LOG\_DEBUG, TAG, \_\_VA\_ARGS\_\_);  
 LOGE("final result:%lf\n", result);  
}

### 调用java 方法

C层调用java层方法流程：

1. 首先获取调用类的jclass对象
2. 通过GetMethodID()获取调用方法的id
3. 是用CallIntMethod()调用方法
4. 如果是引用类型，例如String，则把对应的方法改为CallObjectMethod。

方法说明：

*/\*\*  
 \* 获取方法id  
 \** ***@param*** *clazz 类对象引用  
 \** ***@param*** *name 方法名  
 \** ***@param*** *sig 方法签名  
 \** ***@return*** *返回方法id  
 \*/*jmethodID GetMethodID(jclass clazz, const char\* name, const char\* sig)  
  
*/\*\*  
 \* 调用具体方法  
 \** ***@param*** *obj 类对象引用  
 \** ***@param*** *methodID 方法id  
 \** ***@return*** *具体方法的返回值  
 \*/*jint CallIntMethod(jobject obj, jmethodID methodID, ...)

测试代码：

extern "C"  
JNIEXPORT void JNICALL  
Java\_com\_example\_ndkdemo\_BaseActivity\_callAddMathod(JNIEnv \*env, jobject mainActivitThis) {  
 jclass mainActivitCls = env->GetObjectClass(mainActivitThis);  
  
 // jmethodID (\*GetMethodID)(JNIEnv\*, jclass, const char\*, const char\*);  
 jmethodID addMid = env->GetMethodID(mainActivitCls, "add", "(II)I");  
  
 // jint CallIntMethod(jobject obj, jmethodID methodID, ...)  
 int result = env->CallIntMethod(mainActivitThis, addMid, 1, 1);  
 LOGD("result:%d\n", result)  
  
 // ++++++++++++++++++++++ C调用 public String showString(String str,int value) 函数  
 jmethodID showStringMid = env->GetMethodID(mainActivitCls, "showString", "(Ljava/lang/String;I)Ljava/lang/String;");  
 // jobject (\*CallObjectMethod)(jobject, jmethodID, ...);  
 jstring value = env->NewStringUTF("李元霸");  
 jstring resultStr = (jstring) env->CallObjectMethod(mainActivitThis, showStringMid, value, 9527);  
 const char \* resultCstr = env->GetStringUTFChars(resultStr, NULL);  
 LOGD("r==:%s\n", resultCstr);  
}

### 输出log

输出log的流程：

1. 引入头文件
2. 定义宏
3. 使用宏输出日志
4. 由于宏定义使用了分号，所以具体使用的时候省略分号。

测试代码

// 日志输出  
#include <android/log.h>  
  
#define TAG "Derry"  
  
// \_\_VA\_ARGS\_\_ 代表 ...的可变参数  
#define LOGD(...) \_\_android\_log\_print(ANDROID\_LOG\_DEBUG, TAG, \_\_VA\_ARGS\_\_);  
#define LOGE(...) \_\_android\_log\_print(ANDROID\_LOG\_ERROR, TAG, \_\_VA\_ARGS\_\_);  
#define LOGI(...) \_\_android\_log\_print(ANDROID\_LOG\_INFO, TAG, \_\_VA\_ARGS\_\_);

LOGE("final result:%lf\n", result);

LOGD("result:%d\n", result)

LOGD("r==:%s\n", resultCstr)

### JNI数组操作

在一个局部函数内部，一个函数结束的时候会自动弾栈，自动释放所有的局部成员。所以在一个函数内部，不手动释放相关的资源也是没有问题。但是作为一名C工程师，必须时刻养成用完就释放的习惯，否则很容易出现问题。还有一种可能就是在一个很庞大的方法体内部，如果不主动释放资源，那么这些资源的生命周期将覆盖这个方法的整个执行过程，导致不用的资源长时间占用内存空间。

下面是申请资源和释放资源的成对函数：

GetStringUTFChars：获取字符串UTF的字符

ReleaseStringUTFChars：释放UTF字符

GetIntArrayElements：获取int数组

ReleaseIntArrayElements：释放int数组

GetObjectArrayElement：获取引用类型数组

ReleaseStringUTFChars：释放引用类型数组

获取到C字符串：

const char\* GetStringUTFChars(jstring string, jboolean\* isCopy)

isCopy参数传递NULL表示：NULL等于0，其中0为false表示在本地内部完成翻译转换，不需要开辟Copy机制转换。

ReleaseIntArrayElements()方法会可以把C++修改后的数组刷新给Java层，具体参数说明如下：

*/\*\*  
 \* JNI\_OK 本次C++的修改的数组， 刷新给JVM Java层， 并且释放C++数组  
 \* JNI\_COMMIT 本次C++的修改的数组， 刷新给JVM Java层  
 \* JNI\_ABORT 只释放C++数组  
 \*/*env->ReleaseIntArrayElements(ints, \_ints, JNI\_OK); // 已经释放了

操纵杆原理：

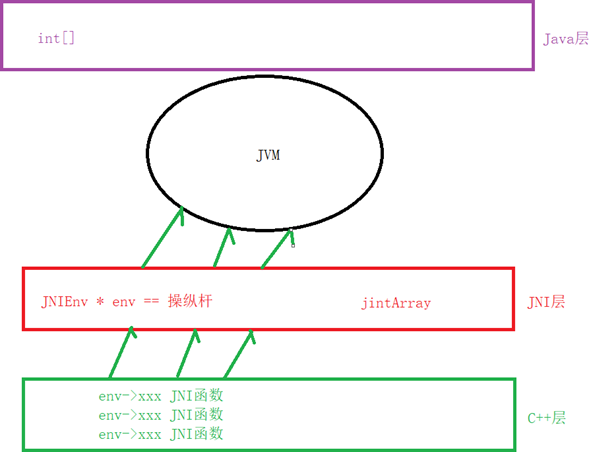
Java和C++交互主要包括：

Java层，

JDK：这个是很广的范围，其中包括jni层，jni含有JNIEnv，这个是Java和C通信的桥梁

C++层

当我们在C++层进行某些操作是，例如获取数组的长度，都需要通过JNIEnv作为中介，所有的操作必须通过JNIEnv（操纵杆）才能完成操作。只有JNIEnv有权利访问到JDK层，JDK层受到影响，Java层才会发生改变。所以这里的目的就是让JNIEnv发送命令到JDK，JDK收到命令进行相关的操作，从而改变Java层的内容或者获取Java相关的信息返回给C++层。可以理解JNIEnv就是操纵杆，C层和Java层的交互都必须通过操纵杆。

可以使用下图描述操纵杆原理  


C++操作数组代码：

extern "C"  
JNIEXPORT void JNICALL  
Java\_com\_example\_ndkdemo\_BaseActivity\_testArrayAction(JNIEnv \*env, jobject thiz, jint count,  
 jstring text\_info, jintArray ints,  
 jobjectArray strs) {  
 int \_count = count;  
 LOGD("C++ \_count:%d\n", count)  
  
 // 非0就是true， 0就是false  
 const char \*\_text\_info = env->GetStringUTFChars(text\_info, NULL); // 0false 在本地内部完成翻译转换，不需要开辟Copy机制转换  
 LOGD("C++ \_text\_info:%s\n", \_text\_info)  
  
 // 作为C工程师，要记住一句话（时时刻刻想着回收你的代码）  
 env->ReleaseStringUTFChars(text\_info, \_text\_info); // 已经释放了  
  
 // *TODO ============================= 下面是 int[] 基本类型的数组* jint \*\_ints = env->GetIntArrayElements(ints, NULL);  
  
 // sizeof(\_ints) / sizeof(int) 你还停留在c工程师的思想  
  
 // jarray父类 jintArray子类 jobjectArray子类 jxxxArray子类  
 int intsLen = env->GetArrayLength(ints);  
 for (int i = 0; i < intsLen; ++i) {  
 // 挪动指针 变量指针 就是 遍历数组  
 LOGI("修改前 C++ ints item:%d\n", \*(\_ints+i))  
  
 \*(\_ints+i) = (i + 100001);  
  
 LOGI("修改后 C++ ints item:%d\n", \*(\_ints+i))  
 }  
  
 // JNI\_OK 本次C++的修改的数组， 刷新给JVM Java层， 并且释放C++数组  
 // JNI\_COMMIT 本次C++的修改的数组， 刷新给JVM Java层  
 // JNI\_ABORT 只释放C++数组  
 env->ReleaseIntArrayElements(ints, \_ints, JNI\_OK); // 已经释放了  
  
 // *TODO ============================= 下面是 引用类型的 数组 Student[] Person[] Test[] String[]* // env->GetIntArrayElements(ints, NULL);  
  
 // jobject GetObjectArrayElement(jobjectArray array, jsize index)  
 // jsize index == 灵感 index 这个是不是 引用类型数组的 下标， 这句话就应该在 for里面 env->GetObjectArrayElement()  
 int strsLen = env->GetArrayLength(strs);  
 for (int i = 0; i < strsLen; ++i) {  
 jstring strItemS = (jstring) env->GetObjectArrayElement(strs, i);  
 const char \* strItemC = env->GetStringUTFChars(strItemS, NULL);  
 LOGI("修改前 C++ strItemC:%s\n", strItemC)  
  
 // 我不用了，就要释放，这个是规范 （养成好习惯）  
 env->ReleaseStringUTFChars(strItemS,strItemC); // 已经释放了  
  
 // env->ReleaseIntArrayElements(ints, \_ints, JNI\_OK);  
 // env.ReleaseObj  
  
 // 寻找灵感  
 jstring updateValue = env->NewStringUTF("Beyond"); // 给上层用  
 env->SetObjectArrayElement(strs, i, updateValue); // 内部会操纵杆刷新  
  
 jstring strItemS2 = (jstring) env->GetObjectArrayElement(strs, i);  
 const char \* strItemC2 = env->GetStringUTFChars(strItemS2, NULL);  
 LOGI("修改后 C++ strItemC2:%s\n", strItemC2)  
  
 // 我不用了，就要释放，这个是规范  
 env->ReleaseStringUTFChars(strItemS2,strItemC2); // 已经释放了  
  
 // 20000行代码  
 // ...  
 } // JNI函数结束，会自动释放，所有的局部成员  
}

### JNI对象操作

获取对象成员方法并调用流程

1. 通过FindClass()获取对象类型的jclass对象
2. 通过GetMethodID()获取方法id
3. 使用CallObjectMethod()调用具体方法
4. 使用ReleaseStringUTFChars()方法释放内存
5. 调用DeleteLocalRef()释放第一步获取的jclass对象。

操作代码如下:

extern "C"  
JNIEXPORT void JNICALL  
Java\_com\_example\_ndkdemo\_BaseActivity\_putObject(JNIEnv \*env, jobject thiz, jobject student,  
 jstring str) {  
 // 最简单的搞定  
 const char \* \_str = env->GetStringUTFChars(str, NULL);  
 LOGD("\_str:%s\n", \_str)  
 // 作为C工程师，要记住一句话（时时刻刻想着回收你的代码）  
 env->ReleaseStringUTFChars(str, \_str); // 已经释放了  
  
 jclass studentClass = env->FindClass("com/example/ndkdemo/jni\_base/Student");  
 jmethodID toStringMethod = env->GetMethodID(studentClass, "toString", "()Ljava/lang/String;");  
 jstring toStringValueS = (jstring) env->CallObjectMethod(student, toStringMethod);  
  
 const char \* toStringValueC = env->GetStringUTFChars(toStringValueS, NULL);  
 LOGD("toStringValueC:%s\n", toStringValueC);  
 env->ReleaseStringUTFChars(toStringValueS, toStringValueC); // 已经释放了  
  
 // setName  
 jmethodID setNameMethod = env->GetMethodID(studentClass, "setName", "(Ljava/lang/String;)V");  
 jstring value1 = env->NewStringUTF("李元霸");  
 env->CallVoidMethod(student, setNameMethod, value1);  
  
 // getName  
 jmethodID getNameMethod = env->GetMethodID(studentClass, "getName", "()Ljava/lang/String;");  
 jstring nameS = (jstring) env->CallObjectMethod(student, getNameMethod);  
 const char \* nameC = env->GetStringUTFChars(nameS, NULL);  
 LOGD("nameC:%s\n", nameC)  
  
 // setAge  
 jmethodID setAgeMethod = env->GetMethodID(studentClass, "setAge", "(I)V");  
 env->CallVoidMethod(student, setAgeMethod, 99);  
  
 // getAge  
 jmethodID getAgeMethod = env->GetMethodID(studentClass, "getAge", "()I");  
 int age1 = env->CallIntMethod(student, getAgeMethod);  
 LOGD("age1:%d\n", age1)  
  
 // void showInfo(String)  
 jmethodID showInfo = env->GetStaticMethodID(studentClass, "showInfo", "(Ljava/lang/String;)V");  
 jstring value2 = env->NewStringUTF("静态的函数 李元霸");  
 env->CallStaticVoidMethod(studentClass, showInfo, value2);  
 env->DeleteLocalRef(studentClass); // 已经释放了

}

### JNI创建对象

创建流程

1. 通过FindClass获取jclass对象
2. 使用AllocObject为上面获取的jclass分配一块内容
3. 通过调用对象的相关方法设置属性
4. 释放内存

注意：

通过AllocObject()方法分配内存时，并不会调用类的构造方法。

通过NewObject()会调用类的构造方法，创建新的对象。

释放内存的2种方式：

1. 使用DeleteLocalRef
2. 使用ReleaseStringUTFChar()

代码如下：

extern "C"  
JNIEXPORT void JNICALL  
Java\_com\_example\_ndkdemo\_BaseActivity\_insertObject(JNIEnv \*env, jobject thiz) {  
 // @Student对象  
 jclass studentClass = env->FindClass("com/example/ndkdemo/jni\_base/Student");  
  
 jobject student = env->AllocObject(studentClass);  
  
 // setName  
 jmethodID setNameMethod = env->GetMethodID(studentClass, "setName", "(Ljava/lang/String;)V");  
 jstring value1 = env->NewStringUTF("刘奋");  
 env->CallVoidMethod(student, setNameMethod, value1);  
 // setAge  
 jmethodID setAgeMethod = env->GetMethodID(studentClass, "setAge", "(I)V");  
 env->CallVoidMethod(student, setAgeMethod, 99);  
  
 jclass personClass = env->FindClass("com/example/ndkdemo/jni\_base/Person");  
 jobject person = env->AllocObject(personClass); // C++ 分配一个对象出来，不会调用此对象的构造函数  
 // env->NewObject(); // C++ 实例化一个对象出来，会调用此对象的构造函数，相当于： new XXX();  
  
 // void setStudent(Student student)  
  
 // void CallVoidMethod(person obj, jmethodID methodID, ...)  
 jmethodID setStudent = env->GetMethodID(personClass, "setStudent", "(Lcom/example/ndkdemo/jni\_base/Student;)V");  
 env->CallVoidMethod(person, setStudent, student);  
  
 // static void putStudent  
 jmethodID putStudent = env->GetStaticMethodID(personClass, "putStudent", "(Lcom/example/ndkdemo/jni\_base/Student;)V");  
 env->CallStaticVoidMethod(personClass, putStudent, student);  
  
 // *TODO 释放工作* // 释放方式一  
 env->DeleteLocalRef(personClass);  
 env->DeleteLocalRef(student);  
 env->DeleteLocalRef(value1);  
 env->DeleteLocalRef(person);  
 // 释放方式二  
 /\*env->GetStringUTFChars()  
 env->ReleaseStringUTFChars()\*/  
}

### JNI全局引用与局部引用理解

在JNI中，即使一个变量定义在函数体外部，它还是一个局部成员。如果需要一个全局的变量，需要使用NewGlobalRef()提升为全局成员。局部成员会随着函数运行结束弾栈而自动释放，全局成员需要手动调用DeleteGlobalRef()方法进行释放，并且把全局成员引用置为空。

jclass dogClass = nullptr; // 在Java的思想中，这个就是全局变量 全局成员，注意了：现在是JNI，这句话 依然是 局部成员  
  
extern "C"  
JNIEXPORT void JNICALL  
Java\_com\_example\_ndkdemo\_BaseActivity\_testQuote(JNIEnv \*env, jobject thiz) {  
 if(!dogClass) { // dogClass == nullptr  
 // dogClass = env->FindClass("com/derry/as\_jni\_2/Dog");  
  
 // 提升全局成员  
 jclass temp = env->FindClass("com/example/ndkdemo/jni\_base/Dog");  
 dogClass = (jclass) env->NewGlobalRef((jobject)temp); // 把 dogClass 提升全局成员  
 env->DeleteLocalRef(temp);  
 }  
  
 // env->NewObject(); // C++ 实例化一个对象出来，会调用此对象的构造函数，相当于： new XXX();  
 jmethodID dogInit = env->GetMethodID(dogClass, "<init>", "()V");  
 jobject dog = env->NewObject(dogClass, dogInit); // Dog dog = new Dog();  
  
 jmethodID dogInit1 = env->GetMethodID(dogClass, "<init>", "(I)V");  
 jobject dog2 = env->NewObject(dogClass, dogInit1, 1); // Dog dog = new Dog(1);  
  
 jmethodID dogInit3 = env->GetMethodID(dogClass, "<init>", "(II)V");  
 jobject dog3 = env->NewObject(dogClass, dogInit3, 1, 2); // Dog dog = new Dog(1, 2);  
}  
  
extern "C"  
JNIEXPORT void JNICALL  
Java\_com\_example\_ndkdemo\_BaseActivity\_delQuote(JNIEnv \*env, jobject thiz) {  
 if (dogClass) { // dogClass != null  
 env->DeleteGlobalRef(dogClass); // JNI函数弹栈，会自动释放所有的“局部成员”，释放后，不会置为NULL  
 dogClass = nullptr;  
 LOGI("手动释放全局引用成功success...")  
 }  
}

## Cmake

### 什么是CMake

在Android Studio 2.2及以上，构建原生库的默认工具是CMake。

CMake是一个跨平台的构建工具，可以用简单的语句来描述所有平台的安装（编译过程）。 能够输出各种各样的makefile或者project文件。CMake并不直接构建出最终的软件， 而是产生其他工具的脚本（如makefile），然后再依据这个工具的构建方式使用。

CMake是一个比make更高级的编译配置工具，它可以根据不同的平台、不同的编译器， 生成相应的makefile或vcproj项目，从而达到跨平台的目的。

Android Studio利用CMake生成的是ninja。ninja是一个小型的关注速度的构建系统。我们不需要关心ninja的脚本，知道怎么配置CMake就可以了。

CMake其实是一个跨平台的支持产出各种不同的构建脚本的一个工具。

### Makefile导入动态库静态库。

Android.mk 要求了解即可，不需要深究

*# 这里面能够决定编译 Login.c Test.c  
  
# 1.源文件在的位置。宏函数 my-dir 返回当前目录（包含 Android.mk 文件本身的目录）的路径。  
# LOCAL\_PATH 其实就是Android.mk文件本身的目录的路径*LOCAL\_PATH := $(call my-dir)  
  
*# LOCAL\_PATH == E:\xuejing\_ndk\02\_VIP\05\_Cmake\shangkeban\_project\ndk28\_mk\app\src\main\ndkDir*$(info "LOCAL\_PATH:======== ${LOCAL\_PATH}")  
  
*# 2.清理*include $(CLEAR\_VARS)  
  
*# TODO 预编译库的引入 == 提前编译好的库*LOCAL\_MODULE := getndk  
  
*# LOCAL\_SRC\_FILES := libgetndk.a # 引入静态库*LOCAL\_SRC\_FILES := libgetndk.so *# 引入动态库  
  
# 预编译静态库的Makeifle脚本 静态库  
# include $(PREBUILT\_STATIC\_LIBRARY)  
  
# 预编译共享库的Makeifle脚本 动态库*include $(PREBUILT\_SHARED\_LIBRARY)  
  
*#引入其他makefile文件。CLEAR\_VARS 变量指向特殊 GNU Makefile，可为您清除许多 LOCAL\_XXX 变量  
#不会清理 LOCAL\_PATH 变量*include $(CLEAR\_VARS)  
# TODO end  
  
*# 3.指定库名字 apk/lib/libMyLoginJar.so 总库  
#存储您要构建的模块的名称 每个模块名称必须唯一，且不含任何空格  
#如果模块名称的开头已是 lib，则构建系统不会附加额外的前缀 lib；而是按原样采用模块名称，并添加 .so 扩展名。*LOCAL\_MODULE := MyLoginJar *# libMyLoginJar.so 总库  
  
#包含要构建到模块中的 C 和/或 C++ 源文件列表 以空格分开*LOCAL\_SRC\_FILES := Login.c *\*Test.c  
  
*# TODO 开始链接进来  
# 静态库的链接  
# LOCAL\_STATIC\_LIBRARIES := getndk  
# 动态库链接*LOCAL\_SHARED\_LIBRARIES := getndk  
  
*# 导入 log  
#LOCAL\_LDLIBS := -llog*LOCAL\_LDLIBS := -lm -llog  
  
*# 4.动态库  
#构建动态库BUILD\_SHARED\_LIBRARY 最后生成总动态库 ---> apk/lib/armeabi-v7a/libMyLoginJar.so*include $(BUILD\_SHARED\_LIBRARY)  
  
*# 静态库  
#构建动态库BUILD\_SHARED\_LIBRARY 最后生成总动态库 ---> apk/lib/armeabi-v7a/libMyLoginJar.a  
# include $(BUILD\_STATIC\_LIBRARY)*

### CmakeList详解

#### 指定Cmake最低支持版本

指定Cmake最低支持版本：

cmake\_minimum\_required(VERSION 3.10.2)

这里并不是指Cmake的最终版本在app build.gradle中指定

externalNativeBuild **{** cmake **{** // path file('src/main/cpp/CMakeLists.txt')  
 path file('src/main/CMakeLists.txt') // 让我们这个生效

version '3.10.2' // Cmake的最终版本  
 **}  
}**

#### 设置工程名

在以前的旧版本，工程名是没有设置的。工程名可以设置，也可以不设置。

project("ndk28\_cmake")

#### 批量导入源文件

可以使用file定义一个变量SOURCE，使用GLOB从源树中手机文件列表，就可以使用\*.cpp \*.c \*.h等文件了。

file(GLOB SOURCE \*.cpp \*.c)

add\_library(native-lib # 库的名字 ---> libnative-lib.so  
 SHARED  
 # cpp的源文件：把cpp源文件编译成 libnative-lib.so 库  
 ${SOURCE}  
 )

#### 添加一个库

添加呀一个动态库使用SHARED关键字，添加一个静态库使用STATIC关键字。添加库名称会自动添加前缀lib，和后缀.so。例如，添加native-lib，系统会自动拼接成libnative-lib.so。

add\_library(native-lib # 库的名字 ---> libnative-lib.so  
 SHARED  
 # cpp的源文件：把cpp源文件编译成 libnative-lib.so 库  
 ${SOURCE}  
 )

#### 查找库

find\_library（）表示查找一个库，怎样知道find\_library里面可以写哪些库呢，可以查看：

D:\android studio\sdk\ndk\21.4.7075529\build\cmake\system\_libs.cmake

文件内容是

这里面有liblog.so，所以find\_library（）就可以写liblog.so。

find\_library(log-abcdafasfsafasf  
 log)

配置了上面的代码之后，查找log库的实际路径是：

D:\Android\Sdk\ndk\21.0.6113669\toolchains\llvm\prebuilt\windows-x86\_64\sysroot\usr\lib\arm-linux-androideabi\16\liblog.so

为什么是21.0.6113669呢，因为local.properties指定了：

ndk.dir=D\:\\android studio\\Sdk\\ndk\\21.4.7075529

如果没有配置ndk.dir，cmake会自动查找合适的版本，并不一定是最新的版本。

为什么是arm-linux-androideabi，因为自己的手机是armeabi-v7a平台的，并且在app的build.gradle配置了：

externalNativeBuild **{** cmake **{** // cppFlags ''  
 // 指定CPU架构是armeabi-v7a 【注意：这里只指定本地库到armeabi-v7a】  
 abiFilters "armeabi-v7a"  
 **}  
}**ndk **{** // 指定CPU架构是armeabi-v7a 【注意：这里只指定编译所有库到armeabi-v7a进apk】  
 abiFilters("armeabi-v7a")  
**}**

为什么是16，因为 minSdkVersion 16：

defaultConfig **{** …  
 targetSdkVersion 30

…

}

通过定义变量和没有定义变量查找库的区别：

定义变量，如果target\_link\_libraries多次也只会查找一次

不定义变量target\_link\_libraries多次也会查找库多次

例如：

下面的代码会查找3次库

target\_link\_libraries(native-lib   
 log  
)

target\_link\_libraries(native-lib   
 log  
)

target\_link\_libraries(native-lib   
 log  
)

下面的代码只会查找一次库：

find\_library(log-abcdafasfsafasf  
 log)  
target\_link\_libraries(native-lib # 被链接的总库  
 ${log-abcdafasfsafasf} # 链接的具体库  
)

target\_link\_libraries(native-lib # 被链接的总库  
 ${log-abcdafasfsafasf} # 链接的具体库  
)

target\_link\_libraries(native-lib # 被链接的总库  
 ${log-abcdafasfsafasf} # 链接的具体库  
)

#### 链接库

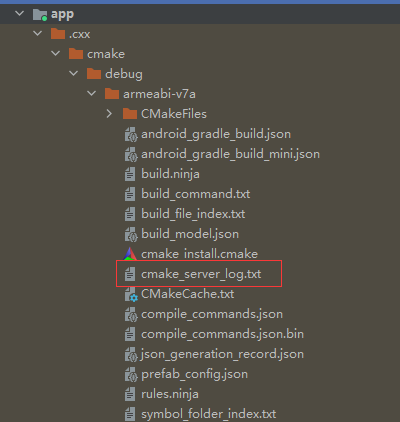
使用target\_link\_libraries连接分库到总库中去。例如下面的代码native-lib是我们的总库，也就是我们在 apk/lib/libnative-lib.so，然后 把log库链接到 总库中去，总库的cpp代码就可以使用 android/log.h的库实现代码了。

target\_link\_libraries(native-lib # 被链接的总库  
 ${log-abcdafasfsafasf} # 链接的具体库  
)

#### log 信息输出的查看

以前的Cmake版本都是在output.txt, 现在最新版本Cmake在metadata\_generation\_stderr.txt或cmake\_server\_log。

在app/.cxx/cmake/debug/armeabi-v7a/cmake\_server\_log.txt



想及时更新你的日志，请安装一次即可 or Refresh Linked C++ Projects，在Build也可以查看，注意：是点击Sync Now 才会看到。

日志级别：

(无) = 重要消息；

STATUS = 非重要消息；

WARNING = CMake 警告, 会继续执行；

AUTHOR\_WARNING = CMake 警告 (dev), 会继续执行；

SEND\_ERROR = CMake 错误, 继续执行，但是会跳过生成的步骤；

FATAL\_ERROR = CMake 错误, 终止所有处理过程；

#### Cmake变量

CMake中所有变量都是string类型。可以使用set()和unset()命令来声明或移除一个变量。

声明变量：set(变量名 变量值)

set(var 666)

引用变量：message 命令用来打印

message("var = ${var}")

移除变量

unset(var)  
message("my\_var = ${var}") # 会取不到值，因为被移除了

#### 声明列表

声明列表：

set(列表名 值1 值2 ... 值N) 或 set(列表名 "值1;值2;...;值N")

set(list\_var 1 2 3 4 5) # 字符串列表呢？ CMake中所有变量都是string类型  
# 或者  
set(list\_var2 "1;2;3;4;5") # 字符串列表呢？ CMake中所有变量都是string类型

message("list\_var = ${list\_var}")  
message("list\_var2 = ${list\_var2}")

#### CMake流程控制-条件命令

以下条件表示true

1. ON，YES，TRUE，Y，非0的值

以下条件表示false

0，OFF，NO，FALSE，N，IGNORE，NOTFOUND

set(if\_tap OFF) # 定义一个变量if\_tap，值为false  
set(elseif\_tap ON) # 定义一个变量elseif\_tap，值为ture  
  
if(${if\_tap})  
 message("if")  
elseif(${elseif\_tap})  
 message("elseif")  
else(${if\_tap}) # 可以不加入 ${if\_tap}  
 message("else")  
# endif(${if\_tap}) # 结束if  
endif() # 结束if 可以不加  
# 注意：elseif和else部分是可选的，也可以有多个elseif部分，缩进和空格对语句解析没有影响

#### CMake流程控制-循环命令

Cmake While循环，使用break()可以跳出整个循环，使用continue()可以继续当前循环，下面的功能表示a等不等于xxx继续循环。

# NOT! a STREQUAL "xxx" (a等不等于xxx，不等于)  
set(a "")  
while(NOT a STREQUAL "xxx")  
 set(a "${a}x")  
 message(">>>>>>a = ${a}")  
endwhile()

Foreach循环:

foreach(item 1 2 3)  
 message("1item = ${item}")  
endforeach(item) # 结束for

区间：

foreach(item RANGE 2) # RANGE 默认从0开始， 所以是：0 1 2  
 message("2item = ${item}")  
endforeach(item)  
  
foreach(item RANGE 1 6 2) # 1 3 5 每次跳级2  
 message("3item = ${item}")  
endforeach(item)

列表

set(list\_va3 1 2 3) # 列表  
# foreach(item IN LISTS ${list\_va3}) # 没有报错，没有循环  
foreach(item IN LISTS list\_va3) # 正规的写法  
 message("4item = ${item}")  
endforeach(item)

#### CMake自定义函数

Cmake的函数使用function定义，endfunction结束函数定义。

function(方法名，参数1，参数2，参数,…)

endfunction(方法名)

方法内置的一些属性：

ARGC：表示传入参数的个数

ARGV0：表示第一个参数，ARGV1、ARGV2以此类推即可

ARGV：表示所有参数

function(num\_method n1 n2 n3)  
 message("call num\_method method")  
 message("n1 = ${n1}")  
 message("n2 = ${n2}")  
 message("n3 = ${n3}")  
 message("ARGC = ${ARGC}")  
 message("arg1 = ${ARGV0} arg2 = ${ARGV1} arg3 = ${ARGV2}")  
 message("all args = ${ARGV}")  
endfunction(num\_method)  
num\_method(10000 2000 3000) # 调用num\_method函数

#### Cmake预编译库与依赖源码方式

下面的没有很细致了解，先做笔记，以后需要详细了解

cmake\_minimum\_required(VERSION 3.10.2) project("ndk28\_cmake")  
  
# 批量导入 cpp c源文件  
file(GLOB SOURCE ${CMAKE\_SOURCE\_DIR}/cpp/\*.cpp ${CMAKE\_SOURCE\_DIR}/cpp/\*.c)  
  
# 添加一个库（动态库SHARED，静态库STATIC）  
add\_library(native-lib # 库的名字 ---> libnative-lib.so SHARED # 动态库  
 # cpp的源文件：把cpp源文件编译成 libnative-lib.so 库  
 ${SOURCE}  
 )  
  
  
  
# *TODO >>>>>>>>>>>>>>>>>>>>>>>> 预编译库（已经有了 xxx.so / xxx.a）的导入方式# TODO 方式一：推荐的方式*#[[  
# 第一步：导入fmod头文件  
include\_directories("${CMAKE\_SOURCE\_DIR}/cpp/inc")  
# 第二步：导入库文件 （方式一）  
set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -  
L${CMAKE\_SOURCE\_DIR}/jniLibsaaa/${CMAKE\_ANDROID\_ARCH\_ABI}")   
# 第三步链接到总库中去  
target\_link\_libraries( # native-lib是我们的总库  
 native-lib # 被链接的总库  
 log # 自动寻找 # 具体的库 链接到 libnative-lib.so里面去fmod # 具体的库 链接到 libnative-lib.so里面去  
 fmodL # 具体的库 链接到 libnative-lib.so里面去  
 )  
]]  
  
# *TODO 方式二：以前更多使用的方式，老程序员使用的方式*#[[  
# 第一步：导入fmod头文件  
include\_directories("${CMAKE\_SOURCE\_DIR}/cpp/inc")   
# 第二步：导入库文件 （方式二）  
add\_library(fmod SHARED IMPORTED)   
set\_target\_properties(fmod PROPERTIES  
 IMPORTED\_LOCATION  
${CMAKE\_SOURCE\_DIR}/jniLibsaaa/${CMAKE\_ANDROID\_ARCH\_ABI}/libfmod.so)  
  
  
add\_library(fmodL SHARED IMPORTED)   
set\_target\_properties(fmodL PROPERTIES  
 IMPORTED\_LOCATION  
${CMAKE\_SOURCE\_DIR}/jniLibsaaa/${CMAKE\_ANDROID\_ARCH\_ABI}/libfmodL.so)  
  
# 第三步链接到总库中去  
target\_link\_libraries( # native-lib是我们的总库  
 native-lib # 被链接的总库  
 log # 自动寻找 # 具体的库 链接到 libnative-lib.so里面去  
 fmod # 具体的库 链接到 libnative-lib.so里面去  
 fmodL # 具体的库 链接到 libnative-lib.so里面去  
 )  
]]  
  
# *TODO >>>>>>>>>>>>>>>>>>>>>>>> 判断静态库还是动态库 (静态库会直接Copy到总库，动态库则不会)*#[[  
# >>>>>>>>>>> 下面代码不参与 判断 start   
# 第一步：导入fmod头文件  
include\_directories("${CMAKE\_SOURCE\_DIR}/cpp/inc")   
# 第二步：导入库文件 （方式二）  
add\_library(fmod SHARED IMPORTED)   
set\_target\_properties(fmod PROPERTIES  
 IMPORTED\_LOCATION  
${CMAKE\_SOURCE\_DIR}/jniLibsaaa/${CMAKE\_ANDROID\_ARCH\_ABI}/libfmod.so)   
add\_library(fmodL SHARED IMPORTED)  
set\_target\_properties(fmodL PROPERTIES IMPORTED\_LOCATION  
${CMAKE\_SOURCE\_DIR}/jniLibsaaa/${CMAKE\_ANDROID\_ARCH\_ABI}/libfmodL.so)   
# >>>>>>>>>>> 上面代码不参与 判断 end  
  
## OFF=0=false ON=1=true # set(isSTATIC OFF)  
set(isSTATIC ON)  
  
if(${isSTATIC})   
 # 导入静态库  
 add\_library(getndk STATIC IMPORTED)  
 # 开始真正导入 静态库   
 System.loadLibrary("getndk"); // 如果是动态库，这里需要加载，否则注释  
 set\_target\_properties(getndk PROPERTIES IMPORTED\_LOCATION  
${CMAKE\_SOURCE\_DIR}/cpp/libgetndk.a) message("isSTATIC == static")  
else(${isSTATIC})   
 # 导入动态库  
 add\_library(getndk SHARED IMPORTED)  
 # 开始真正导入 动态库   
 System.loadLibrary("getndk"); // 如果是动态库，这里需要加载， 否则注释  
set\_target\_properties(getndk PROPERTIES IMPORTED\_LOCATION  
${CMAKE\_SOURCE\_DIR}/jniLibsaaa/${CMAKE\_ANDROID\_ARCH\_ABI}/libgetndk.so) message("isSTATIC == shared")  
  
endif(${isSTATIC})  
  
target\_link\_libraries( # native-lib是我们的总库native-lib # 被链接的总库  
log # 自动寻找 # 具体的库 链接到 libnative-lib.so里面去  
getndk # *TODO 具体的库 链接到 libnative-lib.so里面去【这个库，有可能是静态库，有可能是动态库】*  
fmod # 具体的库 链接到 libnative-lib.so里面去fmodL # 具体的库 链接到 libnative-lib.so里面去  
)  
]]  
  
# rtmp的时候，就全盘采用 源码构建方式  
# *TODO >>>>>>>>>>>>>>>>>>>>>>>> 依赖源码的方式（已经有了 xxxc / xxx.c / xxx.c ...）的导入方式*# 引 入 get 子 目 录 下 的 CMakeLists.txt   
add\_subdirectory(${CMAKE\_SOURCE\_DIR}/cpp/libget)  
# 引 入 count 子 目 录 下 的 CMakeLists.txt   
add\_subdirectory(${CMAKE\_SOURCE\_DIR}/cpp/libcount)  
  
target\_link\_libraries( # native-lib是我们的总库native-lib # 被链接的总库  
 log # 自动寻找 # 具体的库 链接到 libnative-lib.so里面去get # 具体的库 链接到 libnative-lib.so里面去  
 count # 具体的库 链接到 libnative-lib.so里面去  
 )

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5