C++笔记

一.Mac/Linux系统上利用g++对程序进行编译和运行

1.单文件(.cpp)运行

$ g++ -o xxx(执行文件名) xxx.cpp && ./xxx(执行)

🎉 ~

2.带有头文件和库文件的程序

Hypothesis:

hellow.h(头文件或接口文件)、**hellow.cpp**(库文件)、**main.cpp**(客户文件)

·**方法一**：先编译多个cpp源文件，然后直接将生成的.o文件合成一个执行文件

```
$ g++ -c hellow.cpp    //生成hellow.o
$ g++ -c main.pp       //生成main.o
$ g++ -o test hellow.o mian.o
$ ./test
```

🎉 ~

·**方法二**：创建**静态**链接库

```
$ g++ -c hellow.cpp    //生成hellow.o
$ ar rcs libhellow.a hellow.o//将.o文件压缩为.a文件(命名规范:lib+xxx+.a)
$ g++ -o test main.cpp -L. -lhellow //-L.表示链接库在当前文件夹内
或者直接 g++ -o test main.cpp libhellow.a 不要用-L表示库的位置了
$ ./test
```

🎉 ~

注： 链接静态库时可以直接”-l”+xxx,系统自动寻找“libxxx.a”文件

·**方法三**：创建**动态**链接库

```
$ g++ -c hellow.cpp    //生成hellow.o
$ g++ -shared -fPIC -o libhellow.so hellow.o //(命名规范:lib+xxx+.so)
$ g++ -o test main.cpp -L. -lhellow //使用和静态库一样
$ ./test
```

🎉 ~

以上程序在Mac上运行没有问题，在Linux系统下还要小小修改一下：

```
$ g++ -fPIC -c hellow.cpp    //生成hellow.o
$ g++ -shared -o libhellow.so hellow.o //(命名规范:lib+xxx+.so)
```

```
$ g++ -o test main.cpp -L. -lhellow //使用和静态库一样
$ LD_LIBRARY_PATH=./      //有时加了-L. 仍然找不到动态库,可设定一下环境变量
$ ./test
```

🎉 ~

注：若hellow.cpp内容修改，只需要重新编译前两步生成**新的.o文件和新的动态链接库**就行了，第三步不用再将动态库与main链接起来

注：程序照常运行，静态库中的函数**已经**连接到目标文件中了，删除静态库对目标文件没有任何影响，但静态链接库的一个缺点是会浪费很多内存和存储空间，使用了动态链接库就可以避免这个问题，动态库在连接阶段并不把函数代码连接进来，而只是链接函数的一个引用。当最终的函数导入内存开始真正执行时，函数引用被解析，动态函数库的代码才真正导入到内存中，此外，动态函数库的另一个优点是，它可以独立更新，与调用它的函数毫不影响。

二、终端运行可执行文件时的参数传入

只需在主函数**int main(int argc, char **argv)**即可，argc代表输入字符串的个数(按空格计数，包括./test在内)，argv即输入参数组成的字符串矩阵。

例如：$ ./test Hellow! This is a test!

则argc=6，而argv即 ./test
Hellow!
This
is
a
test!

6个字符串组成的矩阵，调用时可用**sscanf(argv[k], ”%lf”, &x)**转换为数字x或其他。(注意：需要<stdio.h>头文件，sscanf 括号内为双引号)

三、头文件

1. 一般以：

```
#ifndefine <标识符>
#define <标识符>
        ..........
        ..........
#endif
```

其中 **<标识符>** 一般为改头文件名的改写，如**Integ.h** , 写成 **_INTEG_H_**

2.**#undef <标识符>** 用于消除前面定义的宏标识符

四、基于NVIDIA CUDA的C++编程

1. 所有含有或者调用__global__、__device__声明函数的文件不再适用.cpp后缀而是采用.cu后缀。

2. 编译.cu文件不再适用gcc或者g++，而是使用cuda自带的nvcc编译器，使用方法同g++编译器。

3. __global__声明的kernel函数如果要调用__device__函数，该__device__函数必须和kernel函数在同一个.cu文件内。

4. 不能在a.cu文件内直接 #include “b.cu” , 这样会使编译错误，因为在编译b.cu时已经定义过一次其中的函数，再编译a.cu时因为include的原因会重新再定义b.cu中的函数，导致重复定义b.cu中的函数。遇到这种情况可以重新建立一个b.h的头文件，头文件中只有函数声明，#include “b.h” 不算重复定义。

5. __device__声明定义的全局变量只在**定义的文件内**有效，cuda编程尽量不要暴露__device__全局变量，可以通过接口函数(内含cudaMemcpyToSymbol)给全局变量赋值。

6. kernel函数中的参数是形式参数，不能直接传址，kernel函数在有数组或指针作为参数时要尤其注意，需要先在device上定义并申请空间。

五、C++工程编译问题

1. 一般c++工程包括的文件夹：
    common文件夹：
        inc文件夹：包含.h头文件
        src文件夹：包含.cpp源文件
        lib文件夹：包含生成的链接库
        data文件夹：包含计算结果
    bin/Debug文件夹：生成的可执行文件
    其他文件夹：包含其他各种文件

2. 编译时可在g++后加上-I(大写i)来指定#include <xx.h>先搜索的文件夹，如：
g++ -o main.cpp -I../inc/　　(..表示上层目录，.表示当前目录)

注意 #include ”xx.h” 优先搜索当前目录，而<>则不会。

六、基于MPI的C++编程

1. 加入 mpi.h 头文件：

```
#include <mpi.h>
```

2. 初始化：

```
MPI_Init(&argc, &argv);
```

3. 获得当前进程的序号：

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

4. 获得总进程：

```
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

5. 结束MPI进程并行：

```
MPI_Finalize();
```

6. 编译使用**mpicc**或者**mpic++**

7. 执行test.exe:

```
mpirun -n 2 ./test.exe
```

注*：MacBook Pro (Retina, 13-inch, Early 2015) 只有2个物理CPU，故执行 test.exe 时 -n 后面的数字不能大于2，故size=2, rank为[0,1]。