

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.cpp //生成main.o
```

```
$ g++ -o test hellow.o main.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：

#ifndef <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]`。

七、结构和链表

1. 创建的 **结构变量** 和 **对象** 都不是 **指针**，除非按 **指针的格式** 创建或者创建 **数组**

eg: `struct date p;` // 创建 `date` 结构变量 `p`, `p` 不是指针

`struct date *q;` // 按指针的格式创建, `q` 是指针;

`struct date m[3];` // 创建 **数组**, 单独的 `m` 是指针

`q = &p;` // `q` 指针指向 `p` 的地址

2. 结构变量的访问：

<1> 结构变量名 . 成员名

<2> (`*` 指向结构的指针) . 成员名 **注: (`*`) 取内容, 括号不能省略**

<3> 指向结构的指针 -> 成员名 (**成员访问运算符 ->**)

eg: struct date p, *q;
cout<< p.year << (*q).month << q->date <<endl;

3. **链表**: 使用数组必须**确定长度**, 而且插入移动删除元素时伴随着**大量元素的移动**, 影响效率。链表就是为了解决这个问题引入的可以**动态**扩大缩小移动储存空间的数据结构。

4. 链表节点:

```
struct node
{
    int data;    // 节点数据
    struct node *next;    // 指向下一节点的指针
};
```

链表节点的定义、生成与释放:

struct node * p; // 定义不能少

<1> p = (struct node *)**malloc(sizeof(node))**; // 生成 **free(p)**; // 释放

或者

<2> p = **new** node; // 生成 **delete p**; // 释放

注意: 使用指针前必须先申请空间, 可以用**malloc(C/C++)**或者**new(C++)**;
malloc函数返回值为任意类型的指针 **void *p**, 通过在前面加 **(struct node *)** **强制类型转化**为 node 结构类型指针, 若要转化为其他变量类型方法类似, 对应使用 **free** 函数释放变量空间。**new** 后面可以用 **()** 赋值或者 **[]** 表明**申请空间大小**, 对应用 **delete** 释放变量空间。

eg: int *a, *b, *c;
a = new int; // 申请空间
b = new int (20); // 申请空间**并赋值**, 即 b = new int; *b = 20;
c = new int [20]; // 申请20个**字节的空间**
delete a, b, c; // **释放变量空间**

5. **链表**的建立:

定义一个函数:

```
struct node * create(int n) {    // 函数返回值为 node 类型的指针
    struct node *head = NULL; // 链表头
    struct node *tail, *newnode; // 链表尾和新的节点
    int x;
    for (int i = 0; i < n; i++) {
        cin >> x;
```

```

newnode = new node; // 新节点申请存放空间
newnode->date = x;   // 新节点赋值
if (head == NULL)
    head = newnode;    // 定义链表头
else
    tail->next = newnode; // 将上一节点的 next 指向新节点
tail = newnode;        // 链表尾更新为现节点
}
tail->next = NULL; // 链表结束
return (head);     // 返回链表头指针
}

```

八、面向对象的C++编程

1. **类** ==> **对象** (类的具体)

2. **对象**的初始化: **构造函数**, 与类同名且不指定函数类型, C++支持构造函数的重载

3. **析构函数**: 在类名前加一个“~”, 不能函数重载, 如果不主动调用, 会在程序结束或者使用该对象结束前被系统自动调用, 用来释放对象。

4. **拷贝初始化构造函数**: 可用于拷贝对象, 只有一个参数, 并且是对该类某个对象的引用, 如果没有声明, 系统就会自动生成一个缺省的拷贝初始化构造函数, 拿一个Circle类举例: `Circle(Circle &p) { x=p.x; y=p.y; }`

5. **常成员函数**: C++主张将所有**不修改对象成员**的函数定义为常成员函数, 只有常成员函数可以操作常对象。

eg: `int funcName() const { return 0; }`

6. **常数据成员**: 在创建对象时**初始化后不再被修改**的数据成员, 可以声明为**const**, 使其受到强制保护, 构造函数只能通过**初始化列表**对其进行初始化。

eg:

```

int c;
const int a, b;    // 常数据成员的声明
Circle(int r, int m, int n) : a(m), b(n) // 初始化列表初始化常数据成员
{
    c = r;
}

```

7. **静态成员**: 静态成员是该类的所有对象共享的成员, **初始化在类外进行**, 在内存中只储存一次。

eg: **static** int num; // 在类的定义内声明为静态成员
int **Circle::**num = 10; // 在类外初始化, 要加**作用域运算符::**

8. **静态成员函数**: 也是该类所有对象共享的成员, 而不是某个对象的成员。可以直接调用静态数据成员, 但是不能直接调用非静态数据成员, 而是要**通过某个对象来调用**。

eg: **static** int num;
static void funcName(Circle p) { cout<<p.x<<p.y<<endl; }// **通过对象**
static int getn() { return num; } // 直接调用静态数据成员

主函数中未创建对象, 便可**直接调用**静态成员函数。

eg: int **Circle::**num = 10;
cout<<**Circle::**getn()<<endl; // 主函数内未创建对象要加**作用域运算符::**

9. **友元函数**: 非成员函数访问类的私有数据成员的方法, 在类内声明, 但是在**类外给出具体实现**。

eg: **friend** double funcName(Circle p1, Circle p2); // 类内声明友元函数
double funcName(Circle p1, Circle p2) // **类外不加 friend 标识**
{ return (p1.x+p2.x); } // 可以访问对象的私有成员

10. **基类** ==> **派生类**
父类 ==> **子类**

eg: class Cylinder : public Circle {...}; // 公有继承
各种派生类的继承关系复杂, 使用时自行查阅。

11. **派生类的构造函数**: 类似于常数据成员的**初始化列表**, 调用基类的构造函数

eg: Circle(int a, int b){ x=a; y=b; } //基类的构造函数
Cylinder(int r, int m, int n) : **Circle(m, n)** { z=r }//调用基类构造函数

注意: 注意若是**派生类**中还有**基类子对象** (即**派生类**的数据成员中包含**基类对象**), 则还要在**初始化列表**后加上**子对象**的初始化。

eg: // **派生类 Cylinder** 的数据成员中包含一个**基类 Circle 对象 A**
private:
 Circle A;
public:
 Cylinder(int r, int m, int n, int p, int q) : Circle(m, n), **A(p, q)**

{ z = r; } // 初始化列表后加上子对象的初始化, 用逗号隔开

12. 先执行派生类的析构函数, 再执行基类的析构函数: 派生类 ==> 基类。

13. 多态性: 面向对象程序设计的重要特征之一, 指同一操作作用于不同的对象会产生不同的结果。

静态多态性: 函数重载 和 运算符重载

动态多态性: 虚函数

14. 运算符重载:

<1> 重载为成员函数:

```
eg: int x;
    Circle() { } // 类的构造函数 (不可少), 为主函数中创建Circle c对象
    Circle(int a) { x=a; } // 类的构造函数的重载
    Circle operator+(Circle p) // 对加法重载, 注意返回值应为Circle对象
    { Circle q(x*x+ p.x*p.x); return q; }
    // 创建对象q作为加法重载后的返回值

    Circle a(10), b(11), c; // 主函数中创建3个Circle对象
    c = a + b; // 调用重载后的加法
```

<2> 重载为友元函数:

```
eg: friend Circle operator+(Circle p1, Circle p2); // 类内声明友元函数
    Circle operator+(Circle p1, Circle p2) // 类外不加 friend 标识
    { Circle q(p1.x*p1.x+ p2.x*p2.x); return q; }
```

15. this 指针: 用于标识调用该成员函数的对象

```
eg: Circle operator++() // 对++重载
    { return (*this); } // 返回调用该函数的对象
    // 注意 this 是个指针, 所以要 * 取内容
```

注意: 由于 this 是指针, 所以在访问对象的成员时使用 -> 符号, java 中也有类似的概念, 但是 java 没有指针, 所以可以直接 this.date 访问对象成员。

16. 虚函数: 某个成员函数在基类中被声明为虚函数, 则说明在其派生类中可能有其他实现方法, 其派生类中的同名函数皆为虚函数。

```
eg: // 在基类 Circle 中声明为虚函数
    virtual void funcName1(int a) { cout<<a<<endl; }
    void funcName2(int b) { cout<<b<<endl; } // 基类普通成员函数
```



```
// 派生类1 Cylinder 中有其他实现方法, virtual 标识可以省略
virtual void funcName1(int a) { cout<<a+1<<endl; }
void funcName2(int b) { cout<<b+1<<endl; } //派生类1普通成员函数

// 派生类2 Sphere 中有其他实现方法, virtual 标识可以省略
virtual void funcName1(int a) { cout<<a+2<<endl; }
void funcName2(int b) { cout<<b+2<<endl; } //派生类2普通成员函数
```

若在类和主函数外定义几个函数:

```
void f1(Circle *p) { p->funcName1(10); p->funcName2(20); }
    // 形参为基类指针
void f2(Circle &p) { p.funcName1(10); p.funcName2(20); }
    // 形参为基类对象的引用
void f3(Circle p) { p.funcName1(10); p.funcName2(20); }
    // 形参为基类对象
```

在主函数中:

```
Cylinder a;
f1(&a); f2(a); f3(a);    // &a 取地址带入指针
Sphere b;
f1(&b); f2(b); f3(a);
```

结果: 11, 20; 11, 20; 10, 20;
12, 20; 12, 20; 10, 20;

从结果可知: 对于普通成员函数 funcName2, 无论形参是什么, 操作的始终是基类的成员函数。对于虚函数 funcName1, 当形参是基类指针或者基类对象的引用时, 操作的是对应派生类的成员函数; 当形参是基类对象时, 操作的仍是基类成员函数。

注意: 这一切都是因为 f1, f2, f3 三个函数声明时使用的是基类 (Circle), 正常情况声明时使用的就是派生类 (Cylinder 或 Sphere), 优先调用派生类的同名普通成员函数, 基类的同名普通成员函数会被隐藏。

对于普通成员函数的调用在编译阶段就确定下来了, 声明用的是基类 (Circle), 所以一直指向基类成员函数。对于虚函数而言, 却是在运行阶段传递参数后才决定调用哪个同名函数, 这种方法称为动态联编。

注意：派生类中的虚函数应与基类中的具有相同的名称，参数个数和类型。只有当虚函数操作的是**指向基类对象的指针**或者是**基类对象的引用**时，才是**动态联编**。当虚函数操作**普通基类对象**时采取的不是动态联编。

17. **纯虚函数：**基类中不给出具体实现，没有实际意义的函数，而将具体实现留给**派生类**去做。

抽象类：至少含有一个**纯虚函数**的类。只能用作**基类**，但不能建立抽象类**对象**。不能用作**参数类型**或者**函数返回值类型**。

```
eg:      // 基类 Circle 中不给出具体实现
         virtual double volume() { return (0); }

         // 子类1 Cylinder 中具体实现方法1， virtual 标识可以省略
         double volume() { return 3.14*R*R*h; }

         // 子类2 Sphere 中具体实现方法2， virtual 标识可以省略
         double volume() { return 3.14*R*R*R*3/4; }
```

主函数中：

```
Circle *p;      // 创建抽象类指针 p，不能创建抽象类对象
Cylinder c;     // 创建圆柱对象 c
Sphere s;       // 创建球对象 s
p=&c;           // 抽象类指针 p 指向圆柱对象 c 的地址
cout << "圆柱体积：" << p->volume() << endl;
p=&s;           // 抽象类指针 p 指向球对象 s 的地址
cout << "球体积：" << p->volume() << endl;
```

注意：* 取内容符号， & 取地址 (引用) 符号。在**声明**时 * 代表**指针**， & 代表**引用**；而在调用函数带入参数时，若 p 代表一个值，&p 代表取 **p 的地址**(= 指针)，若 q 为一个指针，则 ***q** 代表该地址所对应的**值**。

```
eg:  // 声明函数时
     void funcName1(int *a) { }; // 形参代表一个指针
     void funcName2(int &a) { }; // 形参代表 a 的引用
     void funcName3(int a) { };  // 形参代表普通 int

     // 调用函数时
     int p(10);
     int *q;
     q = &p;
```

```
funcName1(&p); // 函数需要一个指针，所以 p 取地址  
funcName1(q); // q 即是一个指针  
funcName2(p); // p 作为引用被调用  
funcName3(*q); // *q 取值被传入
```