20190105

> - 注意：
> - 咱们第一周的目标就是熟悉ubuntu，熟悉shell，熟悉vim的操作
> - 下周会进入C++语言基础阶段，全力进入C++的学习， 打好基础对我们来说，是很重要的。如果C++的基础没打好，对后面的学习 也是影响很大的，尤其是做项目时会比较困难。 今天所学内容比较浅显易懂，但还是建议大家都去实现一下代码， 纸上得来终觉浅，绝知此事要躬行~·
>
> **"C++与C的区别" 的知识点**
>
> - [x] 当天所写代码尽量实现一遍,打包上传作业。

C空间的开辟与C++的空间开辟

```
void *test1()//C空间的开辟
{
    int *pInt = (int *)malloc(sizeof(int));//开辟一个动态空间，这个空间位于堆上,malloc位于stdlib.h头文件中
    memset(pInt, 0, sizeof(int));//把空间清零，已备使用,memset位于string.h头文件中
    *pInt = 10;
    cout << "pInt = " << *pInt << endl;
    cout << "pInt = " << pInt << endl;//这样输出的是pInt的地址
    free(pInt);//释放空间
    return 0;
    //如果不销毁，在使用后return pInt;返回这个空间，就可以接着使用在堆上开辟的这个空间
}
```

```
void test2()//C++的空间开辟
{
    int *pInt = new int(17);
    cout << "pInt = " << *pInt << endl;
    cout << "pInt = " << pInt << endl;
    delete  [] pInt;//释放空间
}
```

结果：

```
kyle@ubuntu:20190105$./a.out
pInt = 10
pInt = 0x5628d4027e70
pInt = 17
pInt = 0x5628d4027e70
kyle@ubuntu:20190105$
```

new/delete 与 malloc的差别：

> **不同点**：

> new/delete 是表达式与 malloc是库函数 new表达式在开辟空间时，只申请空间，不会初始化空间 malloc也可以开辟一个数组空间
>
> **相同点**： new/delete 与 malloc/free 都是用于开辟空间的 必须成对出现，否则就会有内存泄漏的可能
>
> malloc底层实现运用了系统调用

关于内存泄漏、踩踏、野指针的概念：

> ——>**内存泄漏是指**：申请了对空间但是使用之后没有释放（回收）， 因为堆空间是物理内存的一部分，如果长时间不释放之前的空间， 就会导致后期开辟不出新的空间，如有的路由器就有这个现象， 长时间运行后可能要重启
> ——>**内存踩踏是指**：内存重叠，常发生在拷贝过程中，会导致结果不正确 ——>**野指针是指**：定义了指针但是未进行赋值

函数重载

> C++支持函数重载，在函数名相同的时候会根据参数的类型、个数顺序进行改编
>
> ### 默认参数及C编程
>
> 默认参数的使用

```
int add (int x , int y=8, int z=3)//默认参数的额设置必须是从右到左进行
{
    return x + y + z;
}
```

> 使用C风格进行编译时，函数名不能重复
>
> 以下例子是在c++中使用c进行混合编程的方法

```
#ifdef __cpluspluse
extern "C"
{
#endif
int add2(int x, int y)
{
return x + y ;
}
#ifdef __cpluspluse
extern "C"
}// end of extern "C"
#endif
```

```
kyle@ubuntu:20190105$g++ default.cc
kyle@ubuntu:20190105$./a.out
add(a,b) = 8
add(a,b) = 5
add(a,b,c) = 9
add (a + b + c= )9
kyle@ubuntu:20190105$
```

指针和引用

什么是引用？与指针的区别？

> 引用是一个变量的别名
>
> 引用底层实现就是指针 ==》常量指针 引用必须要进行初始化，需要绑定一个实体， 引用一经绑定之后就不能改变指向

```
//引用作为函数的参数进行传递
void swap(int *x, int *y)
{//解引用  间接访问；c常用的方式
    int temp = *x;
    *x = *y;
    *y = temp;
}
void swap2(int &x, int &y)//c++引用的方式
{
    int temp = x;
    x = y;
    y = temp;
}
```

以下为几种错误的引用方式：

```
int * func1 ()
{
    int number = 10;//这种用法也是错误的
    return &number;
}
int & getnumber()
{
    int number = 10;//局部变量值存在于函数的栈空间中，
    return number;//当函数执行完毕后，栈空间已经被销毁了，
                  //所以使用return再去操作她它是没有意义的
                  //所以会出现报警
}
int &getNumber ()
{
    int number = 10;//这种用法也是错误的
    return number;
}
注意：
//不要轻易返回一个堆空间变量的引用，这样做可能会发生内存泄漏(用了之后没回收)
//除非有一个内存回收的策略
```

编译结果：

```
kyle@ubuntu:20190105$g++ reference.cc
reference.cc: In function ‘int* func1()’:
reference.cc:47:6: warning: address of local variable ‘number’ returned [-W
return-local-addr]
  int number = 10;//这种用法也是错误的
      ^~~~~~
reference.cc: In function ‘int& getnumber()’:
reference.cc:52:6: warning: reference to local variable ‘number’ returned
-Wreturn-local-addr]
  int number = 10;//局部变量值存在于函数的栈空间中，
      ^~~~~~
reference.cc: In function ‘int& getNumber()’:
reference.cc:59:6: warning: reference to local variable ‘number’ returned
-Wreturn-local-addr]
  int number = 10;//这种用法也是错误的
      ^~~~~~
```

运行结果：

```
kyle@ubuntu:20190105$./a.out
修改之前:
number: 10
ref: 10

修改之后:
number: 101
ref: 101
&number: 0x7ffc8abdb8ac
&ref: 0x7ffc8abdb8ac
a = 10b= 17
a = 17b= 10
```

强制转换

C中使用 number1 = (int) number2;这种形式

C++中使用number3 = static_cast(number4)；这种形式

如以下例子：

```
1  int test()//C中的强制转换方法
2  {
3      int number1 = 10;
4      double number2 = 17.17;
5      cout << "now , the number1 is : " << number1 << endl;
6      cout << "now , the number2 is : " << number2 << endl;
7      number1 = (int) number2;//强制转换方式1
8  //  number1 = int(number2);//强制转换方式2
9      cout << "after, the number1 is : " << number1 << endl;
10     return 0;
11 }
12 void test1()//C++中的强制转换方法
13 {
14     int number3 = 17 ;
15     double number4 = 17.31;
```

```
16      cout << "now , the number3 is : " << number3 << endl;
17      cout << "now , the number4 is : " << number4 << endl;
18      number3 = static_cast<int>(number4);
19      cout << "after, the number3 is : " << number3 << endl;
20
21      int * pInt = static_cast<int *>(malloc (sizeof(int)));//对指针做转换
22      *pInt = 10;
23      cout << "pInt = " << *pInt << endl ;
24      free(pInt);
25  }
```

结果：

```
kyle@ubuntu:20190105$g++ static.cc
kyle@ubuntu:20190105$./a.out
now , the number1 is : 10
now , the number2 is : 17.17
after, the number1 is : 17
kyle@ubuntu:20190105$
```

```
kyle@ubuntu:20190105$g++ static.cc
kyle@ubuntu:20190105$./a.out
now , the number1 is : 10
now , the number2 is : 17.17
after, the number1 is : 17
now , the number3 is : 17
now , the number4 is : 17.31
after, the number3 is : 17
kyle@ubuntu:20190105$
```

inline 函数与宏定义

常见宏定义

这样没有开销，代码执行效率高

```
1   #define multiply(x,y) ((x)*(y))//这里加上括号是进行保护，防止出错
```

内联函数

1. inline 函数的效果与上面带参数的宏定义效果是一样的
2. 同时还可以进行编译，安全性要比带参数的宏定义高
3. inline 函数不能分成头文件和实现文件，必须放到一起
4. inline 函数尽量不要使用for 循环

```cpp
inline //内联函数
int divide(int x, int y)
{
    return x/y; //比起带参宏定义更方便自定义
}

int main(void)
{
    int a = 8; int b = 4;
    cout << multiply(a, b) <<endl;
    cout << divide(a, b) <<endl;
    return 0;
}
```

结果：

```
kyle@ubuntu:20190105$vim inline.cc
kyle@ubuntu:20190105$g++ inline.cc
kyle@ubuntu:20190105$./a.out
32
2
```

- [x] 预习类和对象的内容（一定要记得预习）