C++代码规范

版　　本	0.0.1
制　定　者	李 川
修　订　者	

规范

在团队项目开发过程中，每个人很容易偏向于自己以往的习惯和风格来编写代码，当项目处于初期时，这不会导致太大的影响；但随着项目的进展，代码库越来越大，里面混杂着各种眼花缭乱的代码书写方式，这不但增加了阅读代码的时间，而且还容易带来混乱和误解。为了在项目整个周期里都能从一而终的保持一致的代码风格，也为了能够更快更好的阅读和理解代码，团队成员就需要按照一定的规范和要求来编写代码。鉴于此，制定该规范。

变量

在变量和一些符号紧邻时，为了便于阅读，在变量与符号之间加上一个空格，比如：

```
int32 value = 0; // =号两边各有一个空格

 if(num > 0 && num < 100)  // > && <这些符号两边都有一个空格
{
    do_something();
}
```

类和结构体的命名应按照首字母大写，单词之间用_（下划线）连接，普通变量应按照全小写，单词之间用_（下划线）连接这种方式来命名，比如：

```
//技能管理类
class Skill_Manager
{
public:
    Skill_Manager();
};

Skill_Manager skill_mgr; //类名大写，变量名小写，单词之间用_分隔
```

如果变量是位于类定义里面的成员变量，则需要加上前缀 m_，这样在一个代码块里可以很明显的区分一个变量是普通变量还是成员变量，比如：

```
//技能管理类
class Skill_Manager
{
public:
    Skill_Manager();
    void init(int32 count);
    int32 m_count;
};


void Skill_Manager::init(int32 count)
{
    if(count == 0)
    {
        return;
    }

    m_count = count;
}
```

函数（包括类的成员方法函数）的命名和普通变量名一样，都是以全小写方式命名，单词之间用_（下划线）分隔，比如：

```
int32 get_max_num(int32 a, int32 b)
{
    if(a > b)
    {
        return a;
    }

    return b;
}
```

在声明指针类型的变量名的时候，如果将指针符号紧挨着类型来书写的话，容易造成误解，比如：

```
char* p, q;
```

这样的声明书写方式很容易让人觉得变量 p 和变量 q 都是字符指针类型，其实按

着 c++语法，上面的声明所表达的意思是变量 p 是一个字符指针类型，但是变量 q 只是一个字符类型。为了避免这样的误解，统一将指针符号紧挨着它所声明的变量，所以上面的声明语句应该改写为：

```
char *p, q;
```

函数参数里如果有指针类型或引用类型的形参变量，为了风格一致性，也需要将指针符号或引用符号紧挨着变量名书写：

```
void get_all_person(std::vector<Person> &person_vec, const Person *self);
```

为了更好理解一个变量所代表的含义，在命名变量名的时候，应该给变量取一个能代表该变量所传递信息的名字，比如：

```
int remain_gold = 0; //代表剩余的金币
time_t last_buy_time = 0; //代表上一次购买的时间
```

大括号

位于 if, while, for 语句后的左大括号要另起一行写，这样，阅读代码的时候，左大括号和右大括号对称显示，比较美观：

```
if(remain_gold < 100)
{
    cout << "剩余金币不足 100" << endl;
} else if(remain_gold < 200)
{
    count << "剩余金币不足 200" << endl;
}

while(cur_count > 0)
{
    do_something();
    --cur_count;
```

```
}
```

头文件

为了防止对一个头文件的重复包含，声明头文件时，在文件起首和结尾分别加上类似下述的宏定义判断：

```
#ifndef __GAME_TIMER__
#define __GAME_TIMER__
.....
.....
.....
#endif
```

在#include 其他头文件的时候，后面的路径不应该包含当前目录和上级目录这样的相对路径，而是应该写成更规范的以某一个根目录为前缀的形式，比如：

```
//不规范的形式：
#include “../gs_timer.h”
#include “../../../base/misc.h”

//规范的形式：
#include “gabriel/core/gs/time/gs_timer.h”
#include “gabriel/base/misc.h”
```

const 修饰符

如果对某一局部变量的操作只限于读取，不会修改该变量的时候，应该在声明变量的时候，加上 const 前缀：

```
const int32 cur_count = get_cur_count();

if(cur_count <= 0)
  {
      //当前次数为 0
}
```

如果类的成员函数不会修改类的内部变量时，也应该对成员函数的声明加上 const 修饰符，比如：

```
class Object_Pool
{
public:
    int32 get_pool_size() const;
};
```

在声明一个函数的时候，假如函数内部不会修改传递进来的实参，那么形参变量就应该声明为 const 变量：

```
void print_person(const std::vector<Person> &person_vec);
```

virtual 关键字

在定义从基类中继承下来的虚函数的时候，虽然不增加 virtual 关键字也能编译通过，但为了让阅读代码的人能一眼看出这是虚函数，子类的虚函数前面也都加上 virtual 关键字。

tab 和空格

不同的编辑器里 tab 键所对应的空格数可能不一样，为了让代码在所有的编辑器中都有统一的缩进效果，书写缩进时，不要使用 tab 键，一律用 4 个空格键代替。

常量、枚举、宏

常量、宏、枚举定义时， 按照全部大写方式来书写，单词之间以_（下划线）

分隔开来，比如：

```
const int32 MAX_POOL_SIZE = 1024 * 1024;

enum FILE_TYPE
{
    FILE_TYPE_EXE,
    FILE_TYPE_OBJ,
    FILE_TYPE_PDF,
};

#define DEBUG 1
#define RELEASE 0
```

多行宏定义

如果一个宏定义里包含多行语句，需要将这些语句全部包括在 do while(0) 块中，比如：

```
#define GET_MAX(a, b) \
do { \
    if(a > b) \
     { \
        return a; \
    } \
    \
    return b; \
} while(0);
```

空行风格

在一个函数代码块中，如果出现 if, while, for 语句时，需要在这些语句的前后都加上一个空行，这样代码看起来美观一些，比如：

```
void proc_logic()
{
    const time_t last_time = get_last_time();
    const time_t begin_time = get_begin_time();
```

```
    if(last_time < begin_time)     //if 语句的前后都有一个空行
     {
        return;
    }

    last_time = now();
}
```

当代码中出现一些导致流程跳转的语句，比如 return, break, continue, goto 时，也应该在这些语句的前面加一个空行，引起阅读代码的人的注意，强调此处有流程转变，比如：

```
void update_per_online_player()
{
    int32 offline_count = 0;
    bool is_empty = false;

    if(player_vec.size() == 0)
     {
        is_empty = true;

        return;              //这个 return 语句的前面有一个空行
    }

    for(int32 i = 0; i != player_vec.size(); ++i)
     {
        Player *player = player_vec[i];

        if(player == NULL)
         {
            ++offline_count;

            continue;          //这个 continue 语句的前面也有一个空行
        }
    }
}
```

简化代码

为了让项目代码更容易阅读和理解，应该在满足业务功能需求的前提下，尽

可能的去掉那些冗余的代码，让代码变得更加简洁，精炼，比如：

```
void check_gold()
{
    if(remain_gold < 100)
    {
        say("不足 100");

        return;
    } else
    {
        say("你已经有了 100");
    }
}
```

上面的代码显得有些啰嗦冗余，应该改为下面的等价形式：

```
void check_gold()
{
    if(remain_gold < 100)
     {
        say("不足 100");

        return;
    }

    say("你已经有了 100");
}
```

仔细观察下面的代码：

```
State* Lian_Meng_Fight::get_state()
{
    if(m_state != NULL)
     {
        return m_state;
    }

    m_state = new State();

    return m_state;
}
```

初看起来，上面的代码好像没有冗余的内容，其实如果能改成下面的形式，则更加的简洁，优美：

```
State* Lian_Meng_Fight::get_state()
{
    if(m_state == NULL)
     {
        m_state = new State();
    }

    return m_state;
}
```