Alliance2023 Algorithm Team 代码规范

Developed by NJUST.Alliance.谭恒宇 2022-10-8

前言

遵守此规范有助于减少出错率，提高调试效率、增强代码可读性和可复用性。

规范与约束

代码

大括号

大括号表示一个相对封闭的代码块，常在`if`、`for`、`switch`等语句后使用。若有必要，可以在任意位置使用大括号，如用来突出和划分一段用来调试的代码，可以方便阅读者的理解。

在大括号内代码较少时建议将前大括号放置于上一行，增强代码的紧凑性。在内容较多时可以单独成行。除非特别简单且目的明确、无需修改的单句语句外，必须使用大括号。

示例：

```
int tmp;
while (std::cin >> tmp)
    a.push_back(tmp);
{ //调试校验和
    int sum = 0;
    for (auto &i : a) {
        printf("%d", i);
        sum += i;
    }
    printf("\nsum:%d", sum);
}
```

缩进

缩进格式：`Tab`制表符缩进，**禁止**采用空格缩进

需要缩进的代码块：

> `if`、`else`、`else if`、`for`、`while`、`do{}while();`、`namespace`、`class`、`struct`、函数、分行的`?:`三目运算符 等

不能缩进的标识：

> `Public:`、`protected:`、`private:`、`case #:`

为什么缩进/不缩进：

缩进通常表示一个新的代码块（特指临时变量新的生存域），而访问标识符和case标签不划分变量的生存域，为避免积极的误导，这些地方不采用缩进。

示例：

```
class C_class {
private:
    //something
protected:
    //something
public:
    //something
};
```

```
if (expresstion1) {
    switch (a[0])
    {
    case 0: {
        //something
        break;
    }
    default:
        break;
    }

    do {
        //something
    } while (true);
    //something
} else if (expreestion2) {
    for (int i = 0; i < 0; ++i) {
        //something
    }
} else {
    while (true) {
        //something
    }
}
```

空行

合理利用空行分割 实现一定子功能的 代码区域，有助于提升可读性。适当在空行插入注释来到更好的分割效果

示例：

```
void C_ore::goeSelect() {
    vector< vector<Point> > contours;
    vector<Point>           Poly;
    vector<Rect>            res;
    vector<int>             res_type;

    findContours(img_pre, contours, RETR_LIST, CHAIN_APPROX_SIMPLE);
    for (int i = 0; i < contours.size(); i++)
        { ... }
```

```
    if (res.empty()) return;
    if (res.size() > 4)
        { ... }
    //corcer
    for (int i = 0; i < res.size(); i++)
        { ... }
    //center
    ...
}
```

空格

单行代码规范参考 **Visual Studio** 默认空格规范，除此以外，您还需在以下情况插入空格

1. 二维、高维向量（确保代码可移植性，加入空格是所有编译器都能接受的格式，否则不少编译器会识别为 >> 右移运算符）

```
vector< vector<Point> > contours;
```

2. 多个 case 的合用：

```
case 4: case 6: { // Rect or 'L'
```

3. #include 后：

```
#include <iostream>
#include <opencv.hpp>
#include "ore.h"
#define CAMERA 1
```

4. 以及任何您觉得加入空格能降低歧义度和提升可读性的地方。

变量

考虑变量使用的范围来确定变量的生存域，切不可滥用全局变量。滥用全局变量会使得代码可移植性和可读性、可修改性低。程序员口中常说的“屎山”代码几乎都存在滥用全局变量的问题。无较有代表性的示例，请读者自行斟酌。

注释

要求头文件里的注释比例大于 40%，即将近一半的地方为注释，作为类封装的头文件抬头必须有类功能、定位、接口的介绍以及开发者、开发时间等信息。

源文件里的比例大于 20%，每个函数声明、复杂代码块、复杂表达式、调用不常用外部接口的地方必须存在解释目的的注释。

注释可以为中文也可以为英文，当注释中存在中文时请注意使用通用的中文文件编码格式，常用的中文编码格式有 UTF-8 、 GB2312 、 Unicode 等。不同软件、设备之间传递代码时注意格式的转换。

示例：

```
#pragma once
#ifndef ORE
#define ORE

/*
创建日期:   2021/12/15
```

```
最后更新:   2021/12/15
开发者:    21 视觉 谭恒宇
类功能:    识别矿石面编号及朝向
版权归属:   南京理工大学Alliance战队
*/
#include <vector>
#include <opencv.hpp>
#include "Template_Shape.h"
#define FRAME_THRESHOLD 10
#define DEBUG 1
#define MATCH_FROM_TEMPLATE 1

using namespace std;
using namespace cv;

class C_ore {
private:
    //original img, HSV、pretreated img
    Mat img_org, img_HSV, img_pre;
    struct ore_info {
        //top left,top right,bottom left,bottom right
        //-1 stands for unkown, 0 for 'L', 1 for Rect
        char tl, tr, bl, br;
        //-1 for unkown, 0 for Nothing, 1 for One-dimensional code, 2 for 'R'
        char middle;
        bool known;
        //ore surface id and orientation
        //id: 0 - top 1 - 'R' 2 - bottom
        //ori: 0~3 - One-dimensional code is up/down/left/right
        //maybe I should use enum here,Well,That's OK
        char id, ori;
        void reset() {
            //reset as unkown
            middle = tl = tr = bl = br = -1;
            known = false;
            id = ori = -1;
        }
        ore_info() { reset(); }
    } info;
    Point ROI_offset;
    //The num of frames has been analyzed, frames num threshold
    int frame_num, frame_thre;

    void pretreat();
    Rect get_ROI_from_org();
    //if the info is enough to know id and ori
    bool ore_known();
    void geoSelect();
    void add_info(Point p, int id);
public:
    C_ore() {
        frame_num  = 0;
        frame_thre = FRAME_THRESHOLD;
```

```
    }
    void analyze_img(const Mat& img_original);
    bool get_ore_info(int& id, int& ori);
    //return inaccurate angle
    bool assess_slope(float &angle);

    Mat& GetImg() { return img_org; }
};

#endif // !ORE
```

命名

变量、函数、类、结构体、联合体、枚举类型等的命名必须体现其意义或功能。尽量做到“见名识意”。尽量使用英文单词或英文单词的缩写而不是拼音来命名，若名称包含多个单词考虑使用大小写（即驼峰法）或下划线 _ 来间隔区分。

示例（遗传算法的种群类）：

```
void variation_default(individual_type&);
individual_type mate_default(const individual_type&, const individual_type&);

class population {
public:
    typedef void (*variation_func_type) (individual_type&);
    typedef individual_type(*mate_func_type) (const individual_type&, const individual_type&);
    typedef double (*fitness_func_type) (individual_type&);

    population() { set_rand_seed(); }
    population(vector< individual_type >);
    virtual ~population() {}
    static void set_rand_seed(unsigned int seed = time(NULL)) { srand(seed); }
    static int randnumInRange(int low, int high);

    int size() { return individuals.size(); }
    //void resize(int n) { individuals.resize(n); }
    void clear() { individuals.clear(); }
    void add_individual(individual_type indiv) { individuals.push_back(indiv); }

    void variation(int index, variation_func_type func = variation_default);
    individual_type mate(int index_A, int index_B, mate_func_type func = mate_default);
    double fitness(int index, fitness_func_type func);
private:
    vector< individual_type > individuals;
    friend void swap(population& A, population& B);
    friend class C_genetic;
};
```

函数

在主源文件不可避免的有较多子函数时，建议将所有函数的声明放置在`main`函数之前，声明放在`main`函数之后，使用 CTRL + 鼠标左键 的方式跳转至函数的实现，有利于代码逻辑的呈现。

类封装时头文件里不实现任何超过一行的函数，被实现在类声明中的函数被编译器默认置为内联`inline`函数，滥用`inline`函数会导致潜在的代码膨胀。头文件对应的源文件里的函数实现应遵循类定义时函数声明的顺序依次实现。

示例（主源文件）：

```cpp
#include <cstdio>
#include "Genetic.h"
#define N 1000
#define M 30
#define T 100000
#define STEP 10
using namespace std;
typedef long long ll;

double fitness(individual_type&);
void show_individual(const char*, individual_type&);
C_genetic gene(fitness);
vector< individual_type > individuals;

int main() {
    srand(time(NULL));
    individuals.resize(N);
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < M; ++j)
            individuals[i].push_back(rand());
    gene.add_group(individuals);

    individual_type winner = gene.best_one();
    show_individual("First generation:", winner);

    for (int t = T; t > 0; t -= STEP) {
        gene.run(STEP);
        printf("In %d G ", T - t + STEP);
        show_individual("Best now:", gene.best_one());
    }

    return 0;
}

double fitness(individual_type& indiv) {...}

void show_individual(const char* info, individual_type& indiv) {...}
```

宏定义

灵活使用宏定义可以实现如调试代码到实际代码的快速切换、使用摄像头与否的快速切换等。不建议将可调参数定义为宏，严禁使用带参的宏定义实现应函数实现的功能（宏定义的行为可能与函数不同，与开发者意图的行为不同且难以排查）。

示例：

```cpp
#include <iostream>
#include <opencv.hpp>
#include "ore.h"
#define CAMERA 1
#define VEDIO 1

using namespace std;
using namespace cv;

C_ore ore;
int id, ori;

int main() {
    Mat img_org;
#if CAMERA == 1
#if VEDIO == 0
    VideoCapture camera(0);
#else
    VideoCapture camera("Resourse/4.mp4");
#endif
#else
    img_org = imread("Resourse/jietu.png");
    resize(img_org, img_org, Size(), 0.5, 0.5);
#endif
    while (true) {
#if CAMERA == 1
#if VEDIO == 1
        waitKey(0);
#endif
        //static VideoWriter output("Resourse/res.avi", \
        //  VideoWriter::fourcc('D', 'I', 'V', 'X'), 20, Size(1280, 720) / 2);
        camera.read(img_org);
        if (img_org.empty()) {
        //  output.release();
            waitKey(0);
        }
        resize(img_org, img_org, Size(), 0.5, 0.5);
#endif
        ore.analyze_img(img_org);
        //output.write(ore.GetImg());
        if (ore.get_ore_info(id, ori)) {...}
        waitKey(1);
    }
    return 0;
}
```

逻辑

注释

代码内注释应使用尽量简洁清晰的语言描述程序的行为目的，其他地方（如函数声名处的注释）应详细充分。

示例（函数声明处详细的注释，对实际开发来说示例注释比较过分，仅作演示）：

```
/** @brief Default constructor
@note In @ref videoio_c "C API", when you finished working with video, release CvCapture structure with
cvReleaseCapture(), or use Ptr\<CvCapture\> that calls cvReleaseCapture() automatically in the
destructor.
 */
CV_WRAP VideoCapture();

/** @overload
@brief  Opens a video file or a capturing device or an IP video stream for video capturing with API Preference

@param filename it can be:
- name of video file (eg. `video.avi`)
- or image sequence (eg. `img_%02d.jpg`, which will read samples like `img_00.jpg, img_01.jpg, img_02.jpg, ...`)
- or URL of video stream (eg. `protocol://host:port/script_name?script_params|auth`)
- or GStreamer pipeline string in gst-launch tool format in case if GStreamer is used as backend
  Note that each video stream or IP camera feed has its own URL scheme. Please refer to the
  documentation of source stream to know the right URL.
@param apiPreference preferred Capture API backends to use. Can be used to enforce a specific reader
implementation if multiple are available: e.g. cv::CAP_FFMPEG or cv::CAP_IMAGES or cv::CAP_DSHOW.

@sa cv::VideoCaptureAPIs
*/
CV_WRAP explicit VideoCapture(const String& filename, int apiPreference = CAP_ANY);
```

初始化

永远初始化在非全局声明或非堆空间中的变量，在类的构造函数中初始化所有数据成员，未熟练掌握语言的程序员容易因未初始化所导致的错误花费大量的调试时间。

断言

熟练并灵活运用断言`assert`能提升代码鲁棒性（即各种输入环境下的适应能力），减少因数组越界等问题造成的异常。

枚举类型

在需要区分多个状态、类型时必须采用枚举类型，不采用枚举类型而采用约定编号的方法很容易导致难以察觉的疏忽。

错误做法示例：

```
//top left,top right,bottom left,bottom right
//-1 stands for unkown, 0 for 'L', 1 for Rect
char tl, tr, bl, br;
//-1 for unkown, 0 for Nothing, 1 for One-dimensional code, 2 for 'R'
char middle;
bool known;
//ore surface id and orientation
//id: 0 - top 1 - 'R' 2 - bottom
//ori: 0~3 - One-dimensional code is up/down/left/right
//maybe I should use enum here,Well,That's OK
char id, ori;
```

宏

重申： 严禁使用宏定义来完成本应函数完成的功能

常用的宏示例：

```
#define SOME_THING 1

#if SOME_THING == 1
// code A
#else // SOME_THING == 0
// code B
#endif

#ifdef SOME_THING
// normal
#else // error when undefine SOME_THING
#error "NOT define SOME_THING"
#endif // SOME_THING

#undef SOME_THING
```

如下两种宏都对头文件有保护作用：

```
#pragma once
//it equals to
#ifndef TEST
#define TEST
//your code here
#endif TEST
```

建议使用第一种形式对头文件进行保护以免宏重名的风险。

命名空间 namespace

禁止在头文件中直接`using namespace`。

若同命名空间的函数调用较少，尽量不直接使用整个命名空间来减少命名冲突的概率，即使不使用整个命名空间，也尽量不要与其中的变量、函数、类等重名。

在源文件中函数使用较多时可以通过直接使用命名空间的方式减少代码的冗余。

引用

合理利用引用的传参以减少数值传递的开销，提升代码效率，不改变原参数的情况下使用`const`修饰符。

示例:

```
individual_type mate_default(const individual_type&, const
individual_type&);
```

指针

不熟悉指针时应尽量避免使用指针，使用指针请一定注意指针安全。

即使熟悉指针，在有不麻烦很多的替选项（如引用）时应优先考虑使用其他方式实现。

在使用`new`、`new[]`和`delete`、`delete[]`对指针进行动态内存分配时应考虑是否可以用带封装和引用计数的类来代理内存分配。

数组和库容器

在元素个数上限不确定时避免使用基础的数组来存储，优先考虑使用如`vector`的STL库容器来存储。

常用的标准库容器有`vector`、`queue`、`deque`、`priority_queue`、`stack`、`list`、`map`、`multimap`、`set`、`multiset`等

其他的数据结构（如分块、线段树、平衡树等）建议使用前届已测试并封装好的类。

函数声明与实现

函数应具有明确的目的性，代表一个相对独立的，完整的功能。函数的功能和目的应在声明处有详细的注释体现。

函数声明可以不包括形参的名字，但必须包含完整的形参类型。

示例:

```
void variation_default(individual_type&);
individual_type mate_default(const individual_type&, const
individual_type&);
```

函数实现需包括形参名，默认参数只能存在声明或没有声明直接实现的函数中。

若函数被封装在`.h`和`.cpp`文件内，则声明在头文件，实现在源文件中。头文件不允许实现任何超过一行语句的函数。可以合理使用函数修饰符优化编译。

作为唯一的例外，模板类的模板方法的实现**必须**在头文件中，普通方法的实现**必须**在源文件中。

函数形参和返回值

函数形参的意义、返回值的意义应在函数声明处有详细的说明。

不可滥用按值传递，成员较多、占用字节大的类作为参数传递时若非迫不得已不应使用按值传递，优先考虑引用传递、指针传递或传递其代理类、句柄类。

部分采用引用计数的类按值传递不会产生较大的内存开销，但是函数内任何尝试对其进行修改的操作都会导致**写时复制**。

异常处理

写程序时应有预见性地加入异常处理机制，在程序运行可能遇到的非致命性错误可通过异常抛出`throw`和接收处理`catch`来维持程序的正常运行，以免真正地引发致命性错误。

在赛场上没人希望车跑一半程序挂了，产生致命性错误、请求了严重的程序退出时可以通过有意识的自我重启、保护线程、看门狗程序或硬件看门狗重启弥补。

多线程和线程安全

线程之间的通信一般通过一个或多个缓冲区和数把线程锁、信号量来实现。若使用多线程编程请一定注意线程安全，任何一个公有的数据都应有一把独立的锁，尽量将数据打包在一个类或结构体中。

类封装

意义

为什么要封装一个类？类是通用的，在有很多可移植的类时编程通常是简单而顺利的。类封装对代码的可读性、可维护性、可移植性都有巨大的贡献。

定位

创建一个类时，类的定位非常重要，类定位应该是尽量脱离某个特定的程序，独立能完成一套算法动作的个体。定位切不可与某个特定的过程、函数、工程有不可分的联系，否则封装类就失去了意义。

接口和封装

类应该提供统一、必要的接口，类封装的意义在于将内部实现的方式隐藏，提供的接口完整而方便。

示例（循环队列类&实现）：

```
template <typename T, int NUM> struct DEQUE {
private:
    T q[NUM]; int head, last;
public:
    DEQUE():head(0), last(0) {}
    void push(T v) {
        if(last == NUM) last = 0;
        q[last++] = v;
    }
    bool empty() { return head == last; }
    void pop() { if(++head == NUM) head = 0; }
    T& front() { return q[head]; }
    T& back() { return q[last - 1]; }
};
```

头文件

头文件参照以下格式

```
#pragma once
/*
Creation Date: 2021/12/24
Latest Update: 2022/10/08
Developer(s): 21 THY
(C)Copyright: NJUST.Alliance - All rights reserved
Class functions:
- function1
- function2
- ...
*/
#include <...>
#define ...

class ... {
private:
    ...
public:
    ...
};
```

若有`#pragma once`，可省略`ifdef`对头文件的保护。

源文件

主源文件

主源文件参考格式如下

```
#include <...>
#include "..."
#define ...
using namespace ...;
typedef ...;

class_type varA, varB;

int main(int argc, char* argv[]) {
#ifdef DEBUG
    ...
#else // no DEBUG
    ...
#endif
    return 0;
}
```

不允许在主源文件`main`函数前有任何函数的实现，`return 0;`是必须的。

其他

其他源文件对格式不做要求，但函数的实现顺序须与声明顺序相同。

视觉工程

Mat

`Mat`类是自带引用计数的类，如果直接让`Mat`类型的`a = b`，修改`a`时`b`中的图形也将被改变，这时应使用`a = b.clone();`，也可利用这个特性实现非引用的函数传参修改，但要求还是要使用引用传参。

Named window

`Namedwindow`可以有多种类型，具体取决于第二个参数，请自行查阅。

Trackbar

建议在一个项目中将所有的`trackbar`创建放在一个函数内

示例：

```
void ParametersInit(...) {
#if PARA_DEBUG == 1
    namedWindow(TRACKBAR_NAME, WINDOW_NORMAL);
    // LightBar Parameters
    AddTrackbar("...", &..., ...);
    AddTrackbar("...", &..., ...);
    ...
    // Armor Parameters
    AddTrackbar("...", &..., ...);
    AddTrackbar("...", &..., ...);
    ...
    // ... Parameters
    AddTrackbar("...", &..., ...);
    AddTrackbar("...", &..., ...);
    ...
#endif
}
```

注意不要重复创建同一个拖动条。

工具性工程

创造合适的工具性代码可以让开发变得便利，如HSV颜色范围检测工程等。合理创造并使用自己的工具而非机械地调试。

代码共享

一些通用库（如`PNP`、`ANN`）在开发、封装好后与组内的队员共享。由此每个人负责不同类的开发减少每个人的工作量、加快开发速度，约定好统一的接口以使用同伴开发的类。

Debug

静态查错

静态查错指不运行程序，通过观察代码、逻辑推理、脑内演算来预测程序的行为是否符合本意。学会并熟练静态查错非常重要。一次性写好没有BUG的工程几乎是不可能的，在开始跑程序之前对整个工程进行一遍静态查错能节省大量的调试时间。

动态查错

动态差错指在程序运行时通过中间变量输出、CPU监视器、内存监视器等手段查找工程中运行与预期不符的代码段。利用好现有的查错工具能很好地帮助您更快了解代码的问题所在。