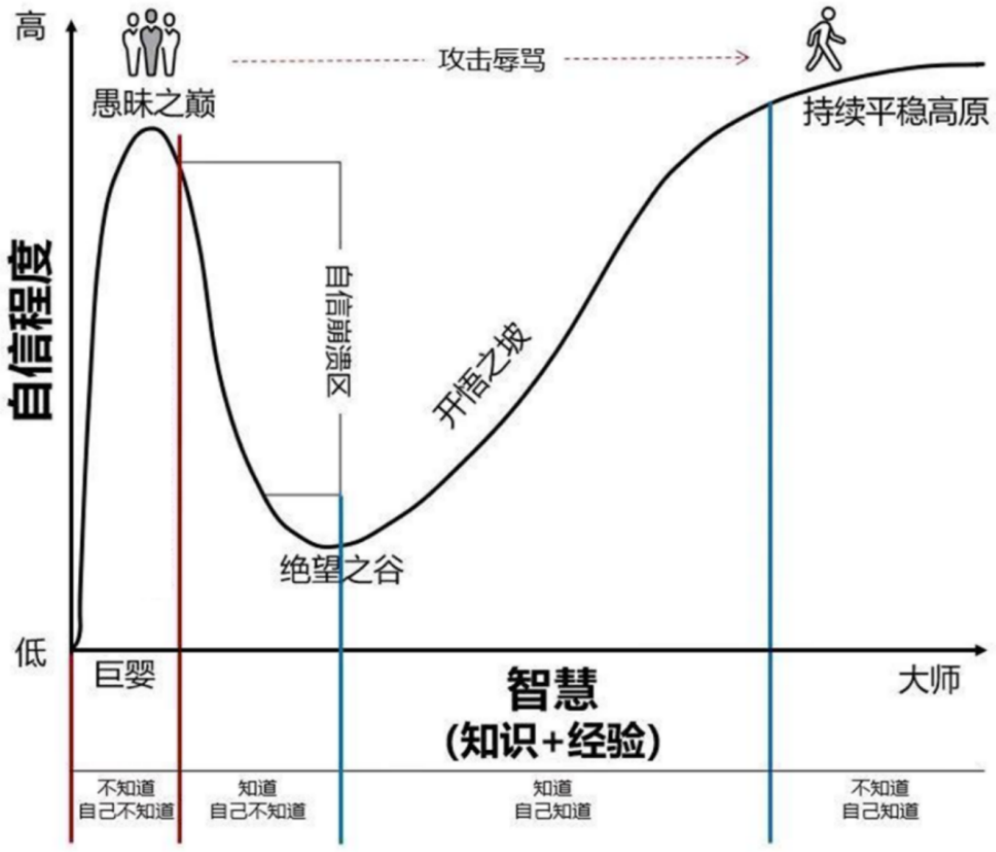
ANP3 Best Practice

目录

前言



借用著名的达克效应，从 “精通c++” 到苦苦经受 core 的毒打，中间可能只隔了入职以来的第一个提交；

想写这样的一个 blog 很久了，虽然我也在 “开悟之坡” 上艰难的攀登，但想尽一些绵薄之力，分享给同学们一些宝贵的经验。

本文主要面向 “绝望之谷” 和 “开悟之坡” 的读者，大师级读者也请不吝赐教，指出文章中的漏洞与不足。

另外，我非常推崇林锐博士的 《高质量C++编程指南》，看本文之前花两三个小时通看一下真的会有非常大的帮助。有余力推荐继续阅读《Effective C++》。

（最新版）高质量C++编程指南.pdf
763KB

最高质量的是谷歌的c++代码规范和Abseil的每周代码建议：https://abseil.io/tips/，里面每周的代码tips都从谷歌的基础代码库和工程实践中总结而来，值得团队每个同学通读，背诵，并反复进行代码练习

和实践，力求做到顺手写出高质量代码要***"像呼吸一样自然"***。

如果要给出非常详细的规则，不如直接看《Effective C++》，本文主要以推荐写法和 case 分析为主。发生在 ANP3 的真人真事，也会逐渐积累成一笔财富。

编码要求

Rule1：

for 循环中不允许出现 `size - 1`

`vector.size()` 的数据类型为 `size_t` 如果这个 `vector` 是空的，`size() - 1` 将会是一个非常大的数，导致越界或无法跳出循环

BadCase:

```cpp
for (size_t i = 0; i < points.size() - 1; ++i) { // 不要不信邪，core 的时候很尴尬
  ...
}
```

GoodCase:

```cpp
for (size_t i = 0; i + 1 < points.size(); ++i) { // 放心食用
  ...
}
```

Rule2：

无论何时，使用指针都要检查有效性。对输入永远保持怀疑，不要相信你的上游，甚至不要相信你自己

环节一旦多起来，函数嵌套够深的情况下，千万不要太自信，对任何输入都要抱有怀疑的心态，做好保护与检查，小心一点总是没错的。检查所消耗的算力能带来更高的鲁棒性，性价比是非常高的。

建议将检查指针养成习惯，不要觉得上边逻辑检查过了，下边就不用检查了，一旦这段代码被复用，曾经的语境被打破就可能会 core

```cpp
void Function1(WorldViewPtr world_view) {
  if (!world_view) {    // 这里只是一种情况，Function1 调用 Function2 的时候不检查输入是没问题的，换了 Function3 就不一定了
```

```
    return;
  }
  Function2(world_view);
}
```

BadCase

C++ | 收起

```
void Function2(WorldViewPtr world_view) {
  // world_view 可能是 nullptr，world_view->chassis 也可能是 nullptr，不要抱有侥幸心理，觉得既然走到 Function2 的逻辑，一定有 world_view 和 world_view->chassis，这种自信可不兴有啊
  const double v0 = world_view->chassis()->speed_limit_mps();
}
```

GoodCase

C++ | 收起

```
void Function2(WorldViewPtr world_view) {
  if (!world_view) { // 养成检查输入的习惯
    return;
  }
  if (world_view->chassis()) { // 不管上下文怎样，我要用一个指针，就一定要检查有效性
    const double v0 = world_view->chassis()->speed_limit_mps();
  }
}
```

Rule3：

不要在一个函数中返回地址

函数中的变量都是临时变量，临时变量的指针指向的内容在函数结束的那一刻就变成了野指针，如果一定要用指针，请使用智能指针。

当然，返回指针的操作整体是不推荐的，如无必要，不要返回指针。

BadCase

C++ | 收起

```
ObstacleInfo* GetVirtualObstacle() {
  ObstacleInfo obstacle_info;
  return &obstacle_info;        // 返回的瞬间，obstacle_info 的内存就被回收了
}
```

C++ | 收起

```cpp
std::vector<ObstacleInfo>& GetObstacles() {  // 返回引用也同理，会 core 的
  std::vector<ObstacleInfo> obs;
  return obs;
}
```

GoodCase

C++ | 收起

```cpp
std::shared_ptr<ObstacleInfo> GetVirtualObstacle() {
  std::shared_ptr<ObstacleInfo> obstacle_info = std::make_shared<ObstacleInfo>();
  return obstacle_info;        // 返回的瞬间，obstacle_info 的 counter 还是 1，不会被回收
}
```

Rule4：

不要相信 vector 中元素的指针

vector 是会自动拓容的，一旦发生拓容，曾经的指针都会失效，非常危险，且难以排查。

BadCase

C++ | 收起

```cpp
std::vector<ObstacleInfo> obstacles;
std::vector<const ObstacleInfo*> interested_obs;
interested_obs.reserve(obstacles.size());
for (const auto& obs : obstacles) {
  if (obs.interested()) {
    interested_obs.push_back(&obs);  // 看似为了减少拷贝，提高了效率，一旦 obstacles 发生拓容操作，interested_obs 中的变量将全部变成野指针
  }
}
```

GoodCase

C++ | 收起

```cpp
std::unordered_map<std::string, ObstacleInfoConstPtr> obs_table;  // 将对象建成 ID 的索引，存储它的智能指针
std::unordered_set<std::string> interested_obs;                   // 只取索引
for (const auto& p : obs_tacle) {
  if (p.second && p.second->interested()) {                       // 所有指针都要检查有效性，参考 Rule2
```

```
    interested_obs.insert(p.first);
  }
}
```

Rule5：

递归函数一定要加层深限制

你永远不知道，下一个死循环是不是出现在你的递归函数里

BadCase

</> C++ | 收起

```
void MyFunc() {
  if (Condition()) {  // 递归函数先写终止条件，养成好习惯
    return;
  }
  MyFunc();           // 常规递归，但很危险!
}
```

GoodCase

</> C++ | 收起

```
const int kMaxDepth = 10000;   // 用 GFlag 或 config 都可以，给一个足够大的数就可以
void MyFunc(int layer) {
  if (layer > kMaxDepth || Condition()) { // 可以增加 log，提示是由于超层深限制而退出
    return;
  }
  MyFunc(++layer);            // 每多一层，layer 增加1，这种写法会多一个参数，但至少不会爆掉
}
```

Rule6：

if 语句中，右值放在 == 的左边

主要是预防手滑

BadCase

</> C++ | 收起

```
ExecutorType executor_type = world_view ? world_view->executor_type() : ExecutorType::DUMMY; // 检查指针！！！
```

```
if (executor_type == ExecutorType::ICA_EXECUTOR) { // == 写成 =, executor_type 就变了，这种 bug 很难查
  ...
}
```

GoodCase

C++ | 收起

```
ExecutorType executor_type = world_view ? world_view->executor_type() : ExecutorType::DUMMY; // 检查指针！！！
if (ExecutorType::ICA_EXECUTOR == executor_type) { // == 写成 = 会编译不通过
  ...
}
```

Rule7：

if 语句中，即使只有一行，也要加 {}

只要你的源码还在，就可能被其它人修改，一行的代码看似简洁，实则非常危险；

BadCase

C++ | 收起

```
if (Condition()) MY_MACRO(a,b,c);  // 没人知道 MY_MACRO 这个宏是几行
```

GoodCase

C++ | 收起

```
if (Condition()) {
  MY_MACRO(a,b,c);  // 虽然不推荐有宏，但这样起码是安全的
}
```

Rule8：

浮点数比较，要考虑精度

BadCase

C++ | 收起

```
double obs_speed_mps;
if (0.0 == obs_speed_mps) { // obs_speed 可能是 9.039581289e-27
  ...
}
```

GoodCase

```cpp
const double kMathEpsilon = 1e-7;
double obs_speed_mps;
if (std::abs(obs_speed_mps) < kMathEpsilon) {
  ...
}
```

Rule9：

即使再不合理的代码，也不要 CHECK

CHECK 只允许出现在初始化阶段，初始化有专门的同学负责，大家一般用不到，所以只要记得不要用 CHECK 就好；

Q：CHECK 可以充分暴露问题，为什么不能用？这样会掩盖问题的。

A：我们现在在做产品！问题的识别不能以 core 为代价。

BadCase

```cpp
if (speed_mps > 3.0e8) {
  CHECK << "Impossible";   // 虽然很合理，但我们开发的是产品，不可以 core!
}
```

GoodCase

```cpp
if (speed_mps > 3.0e8) {
  AERROR << "Impossible speed: " << speed_mps;  // 留给自己一些 log
  // 中间可以加一些异常处理逻辑
  return false;
}
```

Rule10：

默认使用 `std::abs`

`fabs` 和 `abs` 都是 C 函数，在 C++ 标准库中应使用 `std::abs`， `std::abs` 可重载，适用面更广

BadCase

```cpp
if (abs(1.1) > 1.0) {  // abs(1.1) 等于 1，这个条件是进不去的
  // do some thing
}
```

GoodCase

```cpp
if (std::abs(1.1) > 1.0) {  // std::abs(1.1) 等于 1.1，可以正常进入条件语句
  // do some thing
}
```

Rule11：

non-void 函数先写返回值

gcc9 中 non-void 函数不写 `return` 语句是可能编译通过的，但会触发 runtime core

BadCase

```cpp
int BadCase() {
  std::cout << "hello core" << std::endl;
}
```

GoodCase

```cpp
int GoodCase() {
  std::cout << "hello world" << std::endl;
  return 0;   // 刚定义的时候就可以把默认 return 写上，以防忘记
}
```

Rule12：

不要在头文件里写 using namespace xxx 或 using xxx (别名除外);

header 可能会被多隐性包含，头文件中的 using namespace 可能造成命名空间污染，出现问题极其难以排查

BadCase

test.h

```cpp
#include "apn_common/math/vec2d.h"
```

```
#pragma once

namespace apollo {
namespace executor {

using namespace apollo::common::math;          // 这种情况是不允许的，极易造成命名空间污染
using namespace apollo::common::math::lerp;    // 这种也不行，可能造成其它自定义的 lerp 失效

void TestPoints(const std::vector<Vec2d>& points);

}
}
```

GoodCase

C++ | 收起

```
#include "apn_common/math/vec2d.h"

#pragma once

namespace apollo {
namespace executor {

void TestPoints(const std::vector<apollo::common::math::Vec2d>& points);   // 头文件里命名空间补全，手写一遍，确保数据类型的一致性；

// 别名方式的 using 是允许的，它不会造成命名空间污染
class ReferenceLine;
using ReferenceLinePtr = std::shared_ptr<ReferenceLine>;
using ReferenceLineConstPtr = std::shared_ptr<const ReferenceLine>;

}
}
```

Rule13：

当一个类必需持有自身类的时候，禁止使用 std::shared_ptr;

如果自己持有了自己，会造成循环引用，引起内存泄漏；

BadCase

```cpp
class Lane {
 protected:
  std::shared_ptr<Lane> other_lane_; // 设计初衷是想持有其它对象，但存在持有自身的风险
};
```

GoodCase

```cpp
class Lane {
 protected:
  std::weak_ptr<Lane> other_lane_; // 使用 weak_ptr，避免循环引用
  std::string other_lane_id_;      // 或使用 id 索引，避免引用指针
};
```

提升质量 Tips

Recommend1：

提前申请 vector 的空间

知道准确大小就用准确大小，不知道的可以估一个大概的大小，取稍大一点没关系；

```cpp
std::vector<ObstacleInfo> obstacles;
obstacles.reserve(perception_obstacles.size());    // 提前申请空间
for (const auto& pb_obs : perception_obstacles) {
  obstacles.emplace_back(pb_obs);                  // 尽可能使用 emplace_back
}
```

Recommend2：

对于 vector，能用 emplace_back, 就用 emplace_back；

emplace_back 比 push_back 少一次拷贝，积少成多；

```cpp
std::vector<Vec2d> points;
points.reserve(pb_points.size());                        // 提前申请空间
for (const auto& pb_point : pb_points) {
```

```
  obstacles.emplace_back(pb_point.x(), pb_point.y());  // 尽可能使用 emplace_back
}
```

Recommend3：

能用乘法，就不要用除法。

对于计算机而言，乘法是比除法快的；

</> C++ | 收起

```
constexpr double KMH2MPS = 0.277778 // (1.0 / 3.6) = 0.277778
double speed_mps_1 = speed_kmh * KMH2MPS;
double speed_mps_2 = speed_kmh / 3.6;    // 计算量到一定程度的时候，speed_mps_1 是比 speed_mps_2 要快的
```

Recommend4：

类内提前开辟内存

class 并不仅仅可以存属性，一些临时变量，也可以存下来，避免二次开辟内存（PS：效率差距超乎想象，FixedQP 的 P99 要比普通 QP 的 P99 低非常多）

BadCase

</> C++ | 收起

```
class Foo {
 private:
  void Func() {
    std::vector<Vec2d> pionts;
    // 临时构造 pints 将占用很多 CPU 资源
  }
};
```

GoodCase

</> C++ | 收起

```
class Foo {
 private:
  void Func() {
    // 直接使用 points_，效率很高，超乎想象
  }
 private:
  // For pre-malloc
```

```
  std::vector<Vec2d> points_;  // 作为一个成员变量，虽然对业务没什么直接作用，但不用每次都开辟新的内存
};
```

Recommend5：

尽可能使用 math 库里的三角函数

BadCase

C++ | 收起

```
const double cos_theta = std::cos(theta);
```

GoodCase

C++ | 收起

```
apollo::common::math::Angle16 angle = apollo::common::math::Angle16::from_deg(theta);  // 推荐使用 math 库下的 angle
const double cos_theta = apollo::common::math::cos(angle);
// 这里是查表计算，有兴趣可以看下源码，积少成多，大量的三角函数是比较耗时的
```

Recommend6：

for 语句不要超过 3 层

相信函数爆炸的力量，3层 for 下数量级很吓人（PS：地图实属无奈，定好的协议层级太深了）

BadCase

C++ | 收起

```
for (const auto& road : roads) {
  for (const auto& section : road.sections()) {
    for (const auto& lane : section.lanes()) {
      for (const auto& ol : lane.overlaps()) {
        // 写到这里，代码已经非常难读了，效率也堪忧
      }
    }
  }
}
```

Recommend7：

一个函数不要超出150行

150 行并不是一个硬条件，核心思想是不要写太长的函数，可读性会非常低，高可读性的代码意味着高维护性，不想粘在手上放不掉就请把代码写的清晰易懂一些

BadCase

```cpp
void ExportWorkInfo(WorldViewPtr world_view,
std::shared_ptr<::apollo::planning::WorkInfo> work_info) {
  // 此处省略 500 行……，太难读了，可维护性非常差
}
```

GoodCase

```cpp
void ExportWorkInfo(WorldViewPtr world_view,
std::shared_ptr<::apollo::planning::WorkInfo> work_info) {
  // 大函数拆成小函数，一目了然，哪里 Bug 修哪里
  FillSpeedLimit(world_view, adc_lane_info, work_info);
  FillRoadForm(world_view, adc_lane_info, work_info);
  FillCarAhead(world_view, work_info);
  FillLaneChangeStatus(world_view, work_info);
  FillAccessACCToTJA(world_view, work_info);
  FillWTIInformation(world_view, work_info);
  FillODDReminder(world_view, work_info);
}
```

Recommend8：

if 没必要包太多层

还是为了可读性，可读性与可维护性成正比

BadCase

```cpp
void Func(WorldViewPtr world_view) {
  if (world_view) {
    auto loc = world_view->localization_estimate();
    if (loc) {
      // 在这样一个大括号里处理逻辑，可读性比较差
    }
  }
}
```

GoodCase

```C++
void Func(WorldViewPtr world_view) {
  if (!world_view) {
    AERROR << "invalid world_view";
    return;
  }
  if (!world_view->localization_estimate()) {
    AERROR << "invalid localization";
    return;
  }
  // 准入检查做完后，在这里处理逻辑会清晰很多
}
```

Recommend9:

控制传参的数量

传参太多的情况下，会极大的降低函数的可读性

BadCase

```C++
// 10 个参数的函数，真的非常难读！！超级难读！！！
bool do_some_thing(const bool& find_event, const
apollo::executor::RoadEventType& road_event_type,
                   const int& right_of_way, const bool& is_left_virtual,
                   const bool& is_right_virtual, const LaneInfoSegment&
segment,
                   const common::math::Vec2d& segment_xy, const std::string&
refline_name,
                   const std::vector<ReferenceLineInfoPtr>& all_reflines,
ReferenceLineInfoPtr& refline);
```

Recommend10：

如无必要，勿增实体

算法设计阶段切忌过渡设计，宁可设计不足后期再加接口，也不要没有需求创造需求，让算法显得很完整。往往都是实际中的需求前期没识别到，前期设计的接口实际也没人用。更重要的是，前期设计的接口如果冗余或者不合适，将很难删除。（PS:惨痛教训）

Recommend11：

避免 “冲击波” 式代码

控制语句不要嵌套太深、过深的嵌套会严重影响代码的可读性，Ref《 冲击波代码 》

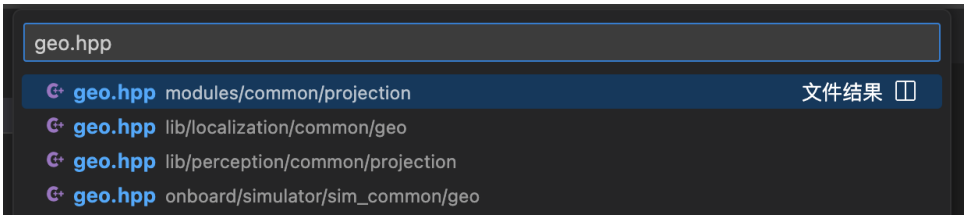
Recommend12：

相同的功能模块抽取成函数

相同的代码存在于多处，对维护成本会非常高，必须对每一处更新进行同步，bug 滋生的概率很高

BadCase

`geo.hpp` 是 andes 内坐标转换的代码，andes 内起码存在 4 份！好在它比较稳定，但这是非常差劲的代码，一旦投影有变，需要四处同步，任何一处未同步都会导致严重 bug。



Recommend13：

对于状态判断函数，充分条件放最后，之前检查必要条件

理解起来太抽象的话可以记为 “不穿插返回”，如逻辑中穿插返回不同的值，可能会漏掉一些逻辑。

BadCase

</> C++ | 收起

```
bool ConditionCheck() {
  if (a) {         // a, b 为递进检查，很可能 a 检查完完全跳过 b、c、d 的检查，易滋生 Bug
    return true;
  } else if (b) {  // 很可能由于命中 b 导致 c 未运行而提前退出
    return false;
  }
  if (c) {
    return true;
  }
  if (d) {
    return false;
  }

  return true;
}
```

GoodCase

</> C++ | 收起

```
bool ConditionCheck() {
  if (a) {              // a、b、c都为必要检查，任一条件不满足即退出
    return false;
  }
  if (b) {
    return false;       // 中途的检查都是返回 false，不穿插返回
  }
  if (c) {
    return false;
  }
  return true;          // 所有必要条件检查完才，达到充分状态才可返回
}
```

真人真事

【Rule2】

在使用指针前没有判空，导致对空指针进行操作了。

对于上游的输入过于自信，理论上少于 2 个点的情况下不可能命中 ReferenceLine 的 Init 函数。但确实发生了。

20230418 ARCF010 core

JIDUL6T79T2E6NP002043-110AD-0601-Routing切段core根因分析

12-28 core 分析

指针未判空Core

Rule4 (配套Recommend1）

根据Recommend1，对于vector这类可变空间容器而言，提前申请好内存空间是非常有必要的，但也应该注意，申请后的空间往往是给定大小，一旦后续再进行添加成员，内存空间必定会重新分配。

20230109-ObstacleInfo访问出core

0919左转core分析

ACC core 复盘汇总--830目标期间core

Rule 9

ACC core 复盘汇总--830目标期间core

Case4:

在while循环里修改逻辑一定要留足break

修改时易忽略外层大逻辑，这样往往会导致死循环

20230505-死循环问题

Rule13：

持有自身类时使用了 std::shared_ptr，没考虑到自己持有自己的情况，造成内存泄漏

内存泄漏根因分析