PCL C++ 编程规范

PCL推荐的命名规范

文件命名

- 所有的文件名单词之间应该用下划线隔开，例如unordered_map.hpp
- 头文件的扩展名为.h
- 模板类实现文件的扩展名为.hpp
- 源文件的扩展名为.cpp

目录命名

- 头文件都应该放在源码目录树中的include/下
- 模板类实现文件都应该放在目录树中的include/impl/下
- 源文件都应该放在目录树中的src/下

Include语句

```
#include <pcl/module_name/file_name.h>
#include <pcl/module_name/impl/file_name.hpp>
#include "file_name.cpp"
```

宏定义命名

例如pcl/filters/bilateral.h对应PCL_FILTERS——BILATERAL_H_。#ifndef和#define定义放在BSD协议后面，代码前面。#endif定义一直在文件结尾，并且加上一句注释掉的宏，对应头文件的宏定义，例如：

```
// the BSD license
#ifndef PCL_MODULE_NAME_IMPL_FILE_NAME_HPP_ // 为避免重复包含头文件而定义的宏
#define PCL_MODULE_NAME_IMPL_FILE_NAME_HPP_
// the code
#endif // PCL_MODULE_NAME_IMPL_FILE_NAME_HPP_
```

命名空间命名

```
namespace pcl_io
{
    ...
}
```

类/结构命名

类名（和其他自定义类型的名称）应该是CamelCased命名规范

```
class ExampleClass;
class PFHEstimation;
```

函数/成员函数命名

```
int
applyExample(int example_arg);
```

变量命名

变量的命名应该单词之间用下划线隔开，例如

```
int my_variable;
```

- 迭代子变量命名 应该反映出它迭代的对象

```
std::list<int> pid_list;
std::list<int>::iterator pid_list; // 指示迭代的对象为点的索引
```

- 常量命名 变量的名字应该是全大写

```
const static int MY_CONSTANT = 1000;
```

- 成员变量名 类的成员变量名命名单词之间用下划线隔开并且以下划线结尾

```
int example_int_; // 对阅读PCL源码很有帮助，可明显区分成员变量与局部变量
```

return语句

return语句需要在圆括号中设定返回值

```
int
main()
{
    return (0);
}
```

pcl推荐的缩进与格式

命名空间缩进格式

在头文件里，命名空间的内容应该缩进两个空格，例如：

```
namespace pcl
{
class Foo
  {
    ...
```

```
  };
}
```

类格式

一个模板类的模板参数必须与类定义在不同行，例如：

```
template <typename T>
class Foo
{
    ...
}
```

函数/类成员函数格式

每一个函数的返回类型声明必须与函数声明放在不同的行，例如：

```
void
bar();
```

在函数实现的时候也是一样，返回类型声明必须与函数声明放在不同的行，例如：

```
void
bar()
{
  ...
}
```

或者：

```
void
Foo::bar()
{
  ...
}
```

或者：

```
template<typename T> void
Foo<T>::bar()
{
  ...
}
```

花括号

```
if (a<b)
{
  ...
}
else
```

```
{
  ...
}
```

空格格式

让我们再来强调一次，在PCL中的每一个代码块的标准缩进是两个空格，这里用当个空格来隔开函数/类成员函数名字与其参数列表，例如：

```
int
exampleMethod(int example_arg);
```

如果在头文件内嵌套应用了命名空间名，需要将其缩进两个空格，例如：

```
namespace foo
{
  namespace bar
  {
    void
    method(int my_var);
  }
}
```

类和结构成员采用两个空格进行缩进，访问权限限定(public,private and protected)与类成员一级，而在其限定下的成员则需要缩进两个空格，例如：

```
namespace foo
{
  class Bar
    {
      public:
      int i;
      int j;
      protected:
      void
      baz()
      {
      }
    };
}
```

设计结构

类和应用程序接口

对于PCL的大多数类而言，调用接口(所有public成员)是不含公开成员变量的，而只有采用两种成员方法(不排除有部分类有公开成员)：

- 固定的类型：它运行通过get/set修改或者添加参数以及输入数据
- 实际实现功能的函数，例如运算、滤波、分割、配准等处理功能

参数传递

get/set类型的方式遵循以下规则：

- 如果大量的数据需要传递，优先采用boost共享指针，而不是传递实际的数据
- 成对的get与set类型成员函数总是需要采用一致的数据类型
- 对于get类型成员函数而言，如果只有一个参数需要被传递则会通过返回值，如果是两个或两个以上的参数需要传递，则通过引用方式进行传递

对于运算、滤波、分割等类型的参数遵循以下规则：

- 无论传递数据的大小，返回参数最好是非指针型参数
- 总是通过引用方式来传递输出参数

如何编写新的PCL类

略

PCL已有点类型介绍和增加自定义的点类型

为什么用PointT类型

```
struct PointXYZ
{
    float x;
    float y;
    float z;
    float padding; // 作为填补位数以满足存储对齐要求
};
```

PCL中有哪些可用的PintT类型

PointXYZ------成员变量：float x,y,z;

PointXYZ是使用最常见的一个点云数据类型，因为它只包含三维XYZ坐标信息，这三个浮点数附加上一个浮点数来满足存储对齐，用户可以利用points[i].data[0]或者points[i].x访问点的x坐标值。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
```

PointXYZI------成员变量：float x,y,z;

PointXYZI是一个简单的XYZ坐标加intensity的point类型，理想情况下，这四个变量将新建单独一个结构体，并且满足存储对齐，然而，由于point的大部分操作会把data[4]元素设置成为0或者1(用于变换)，**不能让intensity与XYZ在同一个结构体中**，如果这样的话其内容就会被覆盖掉。例如，两个点的点积会把他们的第4个元素设置成为0，否则该点积就没有意义，等等。因此，对于兼容存储对齐用3个额外的浮点数来填补intensity，这样在存储方面效率较低，但是符合存储对齐要求，运行效率较高。

```
union
{
    float data[4];
    struct
    {
    float x;
    float y;
    float z;
    };
};

union
{
    struct
    {
        float intensity;
    };
    float data_c[4];
};
```

PointXYZRGBA------成员变量：float x,y,z;uint32_t rgba;

除了RGBA信息被包含在一个整型变量中，其他的和PointXYZI类似。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};

union
{
    struct
    {
        uint32_t rgba;
    };
    float data_c[4];
};
```

PointXYZRGB------float x，Y，z，rgb;

除了RGB信息被包含在一个浮点型变量中，其他和PointXYZRGBA类似。RGB数据被压缩到一个浮点数里的原因在于早期PCL是作为ROS项目的一部分来开发的， 那里RGB数据是用浮点数来传送的，PCL设计者希望所有遗留代码会重新更改(在PCl_ 2. x中很可能这样做)，可能取消此数据类型。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};

union
{
    struct
    {
        float rgb;
    };
    float data_c[4];
}
```

PointXY------float x,y;

简单的二维x-y point结构代码如下：

```
struct
{
    float x;
    float y;
}
```

InterestPoint------float x,y,z,strength;

除了strength表示关键点的强度的测量值，其他的和PointXYZI类似。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
```

```
    struct
    {
        float strength;
    };
    float data_c[4];
}
```

Normal------float normal[3],curvature;

另一个最常用的数据类型，Normal结构体表示给定点所在样本曲面上的法线方向，以及对应曲率的测量值(通过曲面块特征值之间关系获得---查看NormalEsti-mation类API以便获得更多信息，后续章节有介绍),由于在PCL中对曲面法线的操作很普遍，还是用第4个元素来占位，这样就兼容SSE和高效计算，例如，用户访问法向量的第一个坐标，可以通过points[i].data_n[0]或者points[i].normal[0]或者points[i].normal_x，在此再一次强调，曲率不能被存储在同一个结构体 中，因为它会被普通的数据操作覆盖掉。

```
union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;
        float normal_z;
    };
};
```

PointNormal------float x,y,z;float normal[3],curvature;

PointNormal是存储XYZ数据的point结构体，并且包括采样点对应法线和曲率。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;
        float normal_z;
    };
};
```

```
union
{
    float curvature;
    struct
    {
        float data_c[4];
    }
}
```

PointXYZRGBNormal------float x,y,z,rgb,normal[3],curvature;

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};

union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;
        float normal_z;
    };
};

union
{
    struct
    {
        float rgb;
        float curvature;
    };
    float data_c[4];
};
```

PointXYZINormal------float x,y,z,intensity,normal[3],curvature;

PointXYZINormal存储XYZ数据和强度值的point结构体，并且包括曲面法线和曲率。

```
union
{
    float data[4];
    struct
    {
```

```
        float x;
        float y;
        float z;
    };
};
union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;
        float normal_z;
    };
}
union
{
    struct
    {
        float intensity;
        float curvature;
    };
    float data_c[4];
};
```

PointWithRange------float x,y,z(union with float point[4]),range;

PointWithRange除了range包含从所获得的视点到采样点的距离测量值之外，其他与PointXYZI类似。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};

union
{
    struct
    {
        float range;
    };
    float data_c[4];
};
```

PointWithViewpoint------float x,y,z,vp_x,vp_y,vp_z;

PointWithViewpoint除了vp_x,vp_y和vp_z以三维点表示所获得的视点之其他与PointXYZI一样。外

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    struct
    {
        float vp_x;
        float vp_y;
        float vp_z;
    };
    float data_c[4];
};
```

MomentInvariants------float j1，j2，j3;

MomentInvariants是一个包含采样曲面上面片的3个不变矩的point类型，描述面片上质量的分布情况。查看MomentInvariantsEstimation以获得更多信息。

```
struct
{
    float j1, j2, j3;
};
```

PrincipalRadiiRSD------float r_min,r_max;

PrincipalRadiiRSD是一个包含曲面块上两个RSD半径的point类型，查看RSDEstimation以获得更多信息。

```
struct
{
    float r_min, r_max;
};
```

Boundary------uint8_t boundary_point;

Boundary存储一个点是否位于曲面边界上的简单point类型，查看BoundaryEstimation以获得更多信息。

```
struct
{
    uint8_t boundary_point;
};
```

PrincipalCurvatures------float principal_curvature[3],pc1,pc2;

包含给定点主曲率的简单point类型。查看PrincipalCurvaturesEstimation以获得更多信息。

```
struct
{
    union
    {
        float principal_curvature[3];
        struct
        {
            float principal_curvature_x;
            float principal_curvature_y;
            float principal_curvature_z;
        };
    };
    float pc1;
    float pc2;
};
```

PFHSignature125------float pfh[125];

PFHSignaturel25包含给定点的PFH(点特征直方图)的简单point类型，查看PFHEstimation以获得更多信息。

```
struct
{
    float histogram[125];
};
```

FPFHSignature33------float fpfh[33];

FPFHSignature33包含给定点的FPFH(快速点特征直方图)的简单point类型，查看FPFHEstimation以获得更多信息。

```
struct
{
    float histogram[33];
};
```

VFHSignature308------float vfh[308];

VFHSignature308包含给定点VFH(视点特征直方图)的简单point类型，查看VFHEstimation以获得更多信息。

```
struct
{
    float histogram[308];
};
```

Narf36------float x, y, z, roll, pitch, yaw; float descriptor[36];

Narf36包含给定点NARF(归一化对齐半径特征)的简单point类型，查看NARFEstimation以获得更多信息。

```
struct
{
```

```
    float x,y,z,roll,pitch,yaw;
    float descriptor[36];
};
```

BorderDescription------int x,y;BorderTraits traits;

BorderDescription包含给定点边界类型的简单point类型，看BorderEstimation以获得更多信息。

```
struct
{
    int x,y;
    BorderTraits traits;
};
```

IntensityGradient------float gradient[3]；

IntensityGradient包含给定点强度的梯度point类型，查看IntensityGradientEstimation以获得更多信息。

```
struct
{
    union
    {
        float gradient[3];
        struct
        {
            float gradient_x;
            float gradient_y;
            float gradient_z;
        };
    };
};
```

Histogram------float histogram[N];

Histogram用来存储一般用途的n维直方图

```
template<int N>
struct Histogram
{
    float histogram[N];
};
```

PointWithScale------float x, y, z, scale;

PointWithScale除了scale表示某点用于几何操作的尺度(例如，计算最近邻所用的球体半径，窗口尺寸等)，其他的和PointXYZI一样。

```
struct
{
    union
    {
```

```
        float data[4];
        struct
        {
            float x;
            float y;
            float z;
        };
    };
    float scale;
};
```

PointSurfel------float x, y, z, normal[3], rgba, radius, confidence, curvature;

PointSurfel存储XYZ坐标、曲面法线、RGB信息、半径、置信区间和曲面曲率的复杂point类型。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;
        float normal_z;
    };
};
union
{
    struct
    {
        uint32_t rgba;
        float radius;
        float confidence;
        float curvature;
    };
    float data_c[4];
};
```

如何在模板类中使用这些PointT类型

由于PCL模块较多，并且是一个模板库，在一个源文件里包含很多PCL算法会减慢编译过程，在撰写本文档的时候，大多数C++编译器仍然没有很好地来优化处理大量模板文件，尤其是涉及优化（-O2 或者-O3）问题的时候。为

了使包含和链接到PCL库的用户代码编译速度提高，我们使用显示的模板实例化，最终编译链接的库包括了所有可能的模板实例---在这些组合中使PCL中已经定义的point类型的所有模板类都能够直接调用，不需要重新编译，这意味着一旦PCL编译成库，任何用户代码都不需要编译模板化代码，这样就加速了用户编译过程。这个是通过在头文件中首先声明了我们的类和方法，再在模板类实现头文件中进行实现，配置在源文件中进行显示的实例化，最后在编译链接时分别实例化。举一个例子:

```
//foo.h
#ifdef PCL_FOO_
#define PCL_FOO_
template<typename PointT>
class Foo
{
    public:
    void
    compute(const pcl::PointCloud<PointT> &input, pcl::PointCloud<PointT> &output);
}
#endif // PCL_FOO_
```

```
// impl/foo.hpp
#ifndef PCL_IMPL_FOO_
#define PCL_IMPL_FOO_
#include "foo.h"
template<typename PointT>void
Foo::compute(const pcl::PointCloud<PointT> &input, pcl::PointCloud<PointT> &output)
{
    output = input;
}
#endif // PCL_IMPL_FOO_
```

上面定义了Foo::compute方法的模板实现，这种定义通常不与用户代码混合。

```
// foo.cpp
#include "pcl/point_types.h"
#include "pcl/impl/instantiate.hpp"
#include "foo.h"
#include "impl/foo.hpp"
// Instantiations of specific point types
PCL_INSTANTIATE(Foo,PCL_XYZ_POINT_TYPES);
```

最后，上面展示了在PCL中是如何进行显式实例化的，宏PCL_INSTANTIATE仅仅检查给定的类型清单并为每一个类型创建对应一个类实例。pcl/include/pcl/impl/instantiate.hpp中有如下代码：

```
//PCL_INSTANTIATE: call to instantiate template TEMPLATE for all
//POINT_TYPES
#define PCL_INSTANTIATE_IMPL(r, TEMPLATE, POINT_TYPE)\
 BOOST_PP_CAT(PCL_INSTANTIATE_, TEMPLATE)(POINT_TYPE)
#define PCL_INSTANTIATE(TEMPLATE, POINT_TYPES) \
 BOOST_PP_SEQ_FOR_EACH(PCL_INSTANTIATE_IMPL, TEMPLATE, POINT_TYPES);
```

PCL_XYZ_POINT_TYPES在这里(在pcl/include/pcl/impl/point_types.hpp中):

```
//Define all point types that include XYZ data
#define PCL_XYZ_POINT_TYPES \
(pcl::PointXYZ) \
(pcl::PointXYZI) \
(pcl::PointXYZRGBA) \
(pcl::PointXYZRGB) \
(pcl::PointNormal) \
(pcl::PointXYZINormal) \
(pcl::PointWithRange) \
(pcl::PointWithViewpoint) \
(pcl::PointWithScale)
```

实际上，如果只是想给pcl::PointXYZ类型实例化Foo这一个实例类，不需要使用宏，而只需要像下面这样简单地做：

```
//foo.cpp
#include "pcl/point-types.h"
#include "pcl/impl/instantiate.hpp"
#include "foo.h"
#include "impl/foo.hpp"
template class Foo <pcl::PointXYZ>;
```

注意:查看David Vandervoorde和Nicolai M. Josuttis所著的C++ Templates --- The Complete Guide，可获得关于显示实例化的更多信息。

如何增加新的PointT类型

为了增加新的point类型，首先需要进行定义，例如:

```
struct MyPointType
{
    float test;
}
```

然后，得确保自己的代码包含了PCL中特定的类/算法类型的模板头文件的实现，它将和新的point类型MyPointType共同使用，例如，想使用pcl::PassThrough。只需要使用下面的代码即可：

```
#include <pcl/filters/passthrough.h>
#include <pcl/filters/impl/passthrough.hpp>
// test rest of the code goes here
```

如果自己的代码是库的一部分，可以被他人使用，需要为自己的MyPointType类型进行显示实例化。 下面的代码段创建了包含XYZ数据的新point类型，连同一个的test的浮点型数据，这样满足存储对齐。

```
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <pcl/io/pcl_io.h>
struct MyPointType //定义点类型结构
{
    PCL_ADD_POINT4D; // 该点类型有4个元素
    float test;
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW // 确保new操作符对齐操作
```

```
} // 强制SSE对齐
POINT_CLOUD_REGISTER_POINT_STRUCT(MyPointType, // 注册点类型宏
    (float,x,x)
    (float,y,y)
    (float,z,z)
    (float,test,test)
    )

int main(int argc, char **argv)
{
    pcl::PointCloud<MyPointType> cloud;
    cloud.points.resize(2);
    cloud.width = 2;
    cloud.height = 1;

    cloud.points[0].test = 1;
    cloud.points[1].test = 2;
    cloud.points[0].x = cloud.points[0].y = cloud.points[0].z = 0;
    cloud.points[1].x = cloud.points[1].y = cloud.points[1].z = 3;
    pcl::io::savePCDFile("test.pcd",cloud);
}
```

PCL中异常处理机制

本节主要讨论PCL在编写和应用过程中如何利用PCL的异常机制，提高程序的稳健性，首先从PCL开发者角度，解释如何定义和抛出自己的异常，最后从PCL使用者角度出发解释用户如何捕获异常及处理异常。

开发者如何增加一个新的异常类

为了增强程序的稳健性，PCL提供了异常处理机制，作为PCL的开发者需要通过自定义异常以及抛出异常，告诉调用者在出现什么错误，并提示其如何处理，在PCL中规定任何一个新定义的PCL异常类都需要继承于PCLException类，其具体定义在文件pcl/exceptions.h中，这样才能够使用PCL中其他和异常处理相关的 机制和宏定义等。

```
/** \class MyException
* \brief An exception that is thrown when I want it.
*/
class PCL_EXPORTS MyException::public PCLException
{
    public:
        MyException (const std::string & error_description,
            const std::string & filename = "",
            const std::string & function_name = "",
            unsigned line_number = 0) throw()
            : pcl::PCLException(error_description, file name, function name, line_ number){}
};
```

上面是一个最简单的自定义异常类，只定义了空的重构函数，但也足以可以完成对一般异常信息的抛出等功能了。

如何使用自定义的异常

在PCL中为了方便开发者使用自定义的异常，定义下面宏定义：

```
#define PCL_THROW_EXCEPTION(ExceptionName, message)
{
    std::ostringstream s;
    s << message;
    throw ExceptionName(s.str(), __FILE__,"",__LINE__);
}
```

在异常抛出时使用就相当简单，添加下面的代码即可完成对异常的抛出:

```
if(my_requirements != the_parameters_ used_)
    PCL_THROW_EXCEPTION(MyException, "my requirements are not met" <<the_parameters_used);
```

如此，通过宏调用就可以实现对异常的抛出，此处抛出的异常包含异常信息、发生异常的文件名以及异常发生的行号，当然这里异常信息可以包含很多信息，主要因 为在宏定义中通过使用ostringstream的对象，开发者可以任意自定义自己的异常信息，例如添加运行过程中一些重要的参数名或变量名以及其值等，这样就给异常捕获者更多有用的信息，方便异常处理。这里需要说明的另一个问题，以下面代码为例:

```
/* * Function that does cool stuff
* \param nb number of points
* \throws MyException
*/ //Doxygen格式的注释，在进行API文档生成时，会把该注释作为帮助信息，与函数说明放在一起。
void
myFunction(int nb);
```

PCL开发者在自定义函数中，如果使用了异常抛出则需要添加Doxygen格式的注释，这样可以在最终的API文档中产生帮助信息，使用者通过文档可以知道，在调用该函数时需要捕获异常和进行异常处理，本例中在用户调用myFunction函数时，就需要捕获处理MyException异常。

异常的处理

作为PCL的使用者来说，为了能更好地处理异常，你需要使用try...catch程序块。此处和其他异常处理基本一样，例如下面实例:

```
// 在这里调用myFunction时，可以捕获异常
try
{
    myObject.myFunction(some_number);
    // 可以添加更多的其他异常捕获语句
}
// 针对try 块不会的MyException异常进行相应的的处理
catch(pcl::MyException& e)
{
    // MyException异常处理代码
}
// 下面一段代码是对任何异常进行捕获处理的
#if 0
catch(exception& e)
{
    //发生异常的处理代码
}
#endif
```

异常的处理与其自身的上下文关系很大，并没有一般的规律可循，此处列举一些最常用的处理方式:

(1)如果捕获的异常很关键那就终止运行。

(2)修改异常抛出函数的当前调用参数，在此重新调用该函数。

(3)抛出明确而对用户有意义的异常消息。

(4)采取继续运行该程序，这种选择慎用。 本小节只是简单对PCL异常处理机制进行简单实例解说，在PCL官方论坛中有关异常处理的讨论比较多，读者可以自行参考。