C++学习笔记

1. C++常见关键字
2. typedef

typedef是一种存储类型关键字，不能与auto、register、extern、static等存储类型关键字

typedef用来定义存储类型的别名，使用的时候和声明某一类类型的变量是一样，而不能简单的理解成宏定义，如：

1. 1. typedef char\* PCHAR;*//定义了 char\*存储类型的别名PCHAR*
2. 2. typedef int ARRAY[10];*//定义了存储10个int型的数组的存储类型的别名ARRAY*
3. 3. typedef void (\*FUNP)(void); *//定义了指向 返回值是void，参数表是(void)的函数指针的存储类型的别名FUNP*

* 用途一：用于多指针变量的定义

由于 char \* a, b;这种情况下b不会被定义成指针变量，所以采用

1. 1. typedef char \* CHARP; CHARP a, b;

的方式就可以解决该问题

* 用途二：解决struct定义时需要带struct关键字的问题

由于在一些低版本的C++代码中，定义struct类型变量需要写struct关键字

1. struct tagPOINT1
2. {
3. int x;
4. int y;
5. };
6. struct tagPOINT1 p1;

因此通过

1. typedef struct tagPOINT
2. {
3. int x;
4. int y;
5. }POINT;
7. POINT p1;

可以简化书写

* 用途三：简化复杂的变量类型名字，比如上述的函数指针类型

1. typedef char \* PSTR;
2. int mystrcmp(const PSTR, const PSTR);

此处const PSTR相当于char \* const，因为const作用在整个指针上，使得指针本身是个常量，构成了指针常量

1. using

* 用法一：使用名字空间

1. using namespace std；

这样在使用std中的cout的时候就不用写std::cout了

* 用法二：使用名字空间中的函数、变量、类等名称

1. using std::cout;
2. using std::endl;

这种用法相当于缩小了用法一，仅在cout和endl的时候不需要写名字空间，std空间中其他的如cin等使用的时候还需要加名字空间

* 用法三：代替typedef来给存储类型起别名（与typedef效果无异），C++11标准新增

1. using func = void (\*) (int, int);
2. C++变量
3. 变量的声明与定义

变量声明：用于向程序表明变量的类型和名字

变量定义：用于为变量分配存储空间，可指定初始值，程序中，定义有且只有一个

带有extern存储类型表明通知编译器变量在其他地方被定义

带有初始化式的必是变量的定义

除非有extern关键字，否则都是变量的定义

1. C++的宏操作
2. struct \_PointXYZ
3. {
4. PCL\_ADD\_POINT4D;
5. PCL\_MAKE\_ALIGNED\_OPERATOR\_NEW
6. };
7. #define PCL\_ADD\_POINT4D \
8. PCL\_ADD\_UNION\_POINT4D \
9. PCL\_ADD\_EIGEN\_MAPS\_POINT4D
10. #define PCL\_ADD\_UNION\_POINT4D \
11. union EIGEN\_ALIGN16 { \
12. float data[4]; \
13. struct { \
14. float x; \
15. float y; \
16. float z; \
17. }; \
18. };
19. #define PCL\_ADD\_EIGEN\_MAPS\_POINT4D \
20. inline pcl::Vector3fMap getVector3fMap () { return (pcl::Vector3fMap (data)); } \
21. inline pcl::Vector3fMapConst getVector3fMap () const { return (pcl::Vector3fMapConst (data)); } \
22. inline pcl::Vector4fMap getVector4fMap () { return (pcl::Vector4fMap (data)); } \
23. inline pcl::Vector4fMapConst getVector4fMap () const { return (pcl::Vector4fMapConst (data)); } \
24. inline pcl::Array3fMap getArray3fMap () { return (pcl::Array3fMap (data)); } \
25. inline pcl::Array3fMapConst getArray3fMap () const { return (pcl::Array3fMapConst (data)); } \
26. inline pcl::Array4fMap getArray4fMap () { return (pcl::Array4fMap (data)); } \
27. inline pcl::Array4fMapConst getArray4fMap () const { return (pcl::Array4fMapConst (data)); }

第8-10行这一部分，仅仅是简单的宏替换而已，先把后面两个宏替换成原来的内容，之后定义新的宏；通过这种方法构造类，挺新颖的

1. C++指针与引用
2. C++智能指针
3. C++标准库与函数
4. main函数

int main (int argc , char\* argv[ ] & char \*\* argv)

argc：argument count，自变量个数，默认为1

argv：argument vector，指针数组，argv的数组中存储着指向char的指针，默认一个为程序的全名

* char \*\*argv

argv是一个指针变量，argv的指向（\*argv）是char \*，也就是argv指向的也是一个指针 ；

\*argv的指向（\*\*argv）是char。

* char \*argv[]

分析：首先argv是一个数组，数组里面的元素是指针变量（char \*），数组里元素指向的是char。

程序编译后，可执行文件是test.exe

D:\tc2>test

这个时候，argc的值是1

但是执行

D:\tc2>test.exe myarg1 myarg2

argc的值是3。也就是 命令名 加上两个参数，一共三个参数

1. cerr函数

std::cerr是ISO C++标准错误输出流，对应于ISO C标准库的stderr

与std::cout不同，ISO C++要求当cerr被初始化后，cerr.flags() & unitbuf非零（保证流在每次输出操作后被刷新），且cerr.tie()返回&cout。 [1] 即cerr默认和cout同步但无缓冲

cout对应于标准输出流，默认情况下是显示器。这是一个被缓冲的输出，可以被重定向。

cerr对应标准错误流，用于显示错误消息。默认情况下被关联到标准输出流，但它不被缓冲，也就说错误消息可以直接发送到显示器，而无需等到缓冲区或者新的换行符时，才被显示。一般情况下不被重定向。

* 重定向，就是:把原来的 cin 从键盘输入 改为从文件输入。把 原来的 cout 向屏幕输出 改为输出到文件。流的重定向吧，一般字符流可以重新定向到任意输入输出，比如串口，文件。当把cout重新定向到文件那1cout<<"hello&nbsp;world";就会在文件里面显示。

1. C++类相关问题
2. C++的类中自带的的默认构造函数会初始化所有成员变量为0（存疑）
3. 在类中使用union数据类型自动产生类成员变量

问题来源：阅读PCL库PointXYZ类的基类的源码的时候发现

问题代码：/code/mycpp/macros\_based\_union\_member\_test

1. #include <iostream>
2. **using** **namespace** std;
3. **struct** \_PointXYZ{
4. **union**{
5. **float** data[4];
6. **struct**{
7. **float** x;
8. **float** y;
9. **float** z;
10. };
11. };
12. **inline** **void** getdatax(){std::cout<<x;}
13. };
15. **int** main()
16. {
17. \_PointXYZ point;
18. point.getdatax();
19. cout<<endl<<point.y;
20. **return** 0;
21. }

可见，其直接使用了union的数据类型就可以在类中产生成员，由此产生了对于union导出的数据类型与导出变量的疑问。

在C++的标准手册中，有如下该行为的解释：形式为union {member-specification}的联合叫做匿名联合;它定义了一个未命名类型的未命名对象。匿名联合的成员规范只能定义非静态数据成员。[注:嵌套类型和函数不能在匿名联合中声明。匿名联合成员的名称应与声明匿名的范围内的任何其他实体的名称不同。**为了进行名称查找，在匿名联合定义之后，将匿名联合的成员视为已在声明匿名联合的作用域中定义**。

百度上有一个答主说：我很惊讶有一个现代编译器仍然允许该构造。它是从C的早期开始的，大约1975年。在那些日子里，结构和联合成员实际上并没有绑定到特定的结构，而是包含作为属性偏离基地址和数据类型。 最终结果是正确使用结构或联合会产生正确的代码，其表达式按预期进行评估。唯一的区别是，使用与该类型无关的指针滥用结构成员不会被标记为错误。我认为没有任何特殊原因可以不强制执行员工K& R暗示未来的编译器希望能够检查这些用途 - 可能只是为了保存16位域中的符号表空间。

1. 类的类型成员

C++的类中可以有三中成员，成员变量、成员函数与成员类型（又称作变量成员、函数成员、类型成员）

* 类型成员

C++的类中使用using或typedef创建的存储类型说明的成员，与其他成员一样存在访问限制，可以使用private和public进行修饰

* 类型成员的定义方法

类型成员使用typedef和using关键字进行定义

1. class Screen {
2. public:
3. typedef std::string::size\_type pos;    *//使用typedef定*
4. using   pos = std::string::size\_type;  *//使用using*
6. private:
7. pos cursor = 0;
8. pos height = 0, width = 0;
9. std::string contents;
10. }

* 使用作用域运算符 :: 来访问静态成员和类型成员

1. Screen::pos hs =10;

* 类模板中的类型成员

假定T是一个模板类型参数，当编译器在实例化之前遇到类似Screen<T>::pos的时候，它不知道这个pos是一个类型成员还是静态成员。所以C++用typename显式表示模板类中的类型成员

1. typename Screen::pos p;*//表示这是个类型成员*
2. C++类无须创建对象即可访问的内容

静态成员函数、静态成员变量、成员类型