古月 C++程序设计代码协定

Coding styles C++ programming for Classical Moon

2005-11-10 Version 1
2010-05-16 Version 2

基本数据类型

包含文件：#include<hgl/DataType.H>

定义：

```
namespace hgl
{
    typedef unsigned char       uchar;      ///< 无符号字符型
    typedef unsigned int        uint;       ///< 无符号整型

    typedef   signed __int8     int8;       ///< 有符号 8 位整型
    typedef unsigned __int8     uint8;      ///< 无符号 8 位整型
    typedef   signed __int16    int16;      ///< 有符号 16 位整型
    typedef unsigned __int16    uint16;     ///< 无符号 16 位整型
    typedef   signed __int32    int32;      ///< 有符号 32 位整型
    typedef unsigned __int32    uint32;     ///< 无符号 32 位整型
    typedef   signed __int64    int64;      ///< 有符号 64 位整型
    typedef unsigned __int64    uint64;     ///< 无符号 64 位整型
};
```

程序入口

任何一个使用《古月 v18》编写的程序，它的入口函数都是 GameMain，形式如下：

```
#include<hgl/hgl.h>

void GameMain(int argc,wchar_t **argv)
{

}
```

输入变量 argc 是命令行输入的参数个数，argv 是 UTF-16LE(又称 UCS-2)编码的参数内容。例如，在命令行下启动名为 GAME.EXE 的游戏程序。

```
C:\>GAME -fightmode
```

当进入 GameMain 函数时，argv 的内容便是“-fightmode”。

名字空间

为不造成名字空间污染，我们推荐将自己的程序全部放入自己的名字空间中。而《古月 v18》所使用的命字空间便是“hgl”。

```
namespace hgl       ///古月游戏开发库所使用的名字空间
{

}
```

另外由于 OpenAL Ee 是在《古月 v18》之前就有的，所以它有它专门的名字空间“openal”

```
namespace openal  ///OpenAL EE 所使用的名字空间
{
}
```

函数命名方式

要保证程序的可读性，函数的命名至关重要。

一个函数的名称，由它所对应功能的英文词汇组合而成，每个单词的第一个字母大写。

例如：加载文件

```
LoadFromFile(const wchar_t *filename)
```

有一种例外的情况，就是这个函数的功能，它与已往已成为业界标准或惯例的函数功能类似或相同，那么我们将推荐使用类似或相同的命名方式：

例如：以 32 位为单位填充一块内存

```
memset32(void *ptr,uint32 val,uint32 count)
```

前后一致统一的命名也是相当重要的,如果一个类的加载数据用 Load,而另一个类的数据加载用 LoadFromFile,就会给使用程序的人带来重复学习记忆的不必要。

变量的值、函数的返回值

也许很多人会认为这是个很无聊的话题，但我们仍将在这里阐述我们的协定。

是否成功：

很多函数的返回值仅有两种情况：成功、不成功。而针对这种函数，推荐使用 bool 型返回类型。顺之使用 true/false 表示成功与不成功。

比较结果：

在 C/C++程序界，似乎有个不成文的定义。在比较两个数据时，如果数据相等就返回 0，小于就返回负值，大于就返回正值。

枚举命名方式

枚举即是一种定义，也是一种声明，如果不使用很好的命名方式，将会造成很严重的名字空间污染。

枚举本身的名称和函数命名方式一至，枚举量的命名采用枚举名称单词首字母缩写做为前缀，但使用小写方式。后面再跟写枚举量本身意义的词汇组合。

例：文件访问模式枚举

```
enum FileOpenMode                   ///文件访问模式枚举
{
   fomCreate          =0x01,        ///<创建一个文件,只写方式
   fomOpenRead        =0x02,        ///<打开一个文件,只读方式
   fomOpenWrite       =0x04,        ///<打开一个文件,只写方式
   fomOpenReadWrite   =0x02|0x04    ///<打开一个文件,可读可写
};
```

结构、联合体、类的命名方式

结构(struct)、联合体(union)、类(class)的命名方式与函数命名方式一致。无特殊要求。

定义与声明的位置

这里所要讲的一点是：如果一个定义或声明它只在当前的范围内有效，那么请定义在当前范围内。

例如：帧(Frame)数据结构

```
class ActiveTexture2D
{
   struct Frame
   {
      int time;
      int width,height;
      void *data;
      int format;
   };
   ...
```

```
class TextureAnim2D
{
   struct Frame
   {
      int time;
      int width,height;
      int index;
   };
   ...
```

以上两个类，都需要定义一个名为帧(Frame)的数据类型，但是它们定义的内容各不相同。这种情况下，如果我们将帧(Frame)的定义放在类的外部的话，我们就需要创建两个不同名称的结构以免发生冲突，但这样也会造成一定的名字空间污染。所以定义在各自的类中，将会是非常明智的选择。

类成员变量命名方式

对于类中的变量，分为私有(private)、保护(protected)、公有(public)、发布(__published)四种。发布(__published)是一种特殊情况，将不在这里进行讨论。

私有变量是仅仅给自己用的，可能需要有意义的命名，也有可能不需要。我们使用小写方式来书写私有变量。保护类型和公有类型的变量不但需要给自己使用，也会给别人使用，所以它一定会需要一个有意义的命名。这里我们使用和函数命名方式一致的方式来定义这些变量，以让它显得比较明显。例：

```
class ActiveTexture2D
{
   int curframe;              //当前帧数

   void SetCurFrame(int);  //设置当前活动的帧

public:

   __property int CurFrame={read=curframe,write=SetCurFrame};    //当前帧数虚拟变量
};
```

这里和函数命名一样有一种例外就是和以往惯例相似的将使用惯例命名。

不要直接写下毫无意义的数字

直接在程序中写下数字是个很不好的习惯，如果这个数字从源代码文件名、函数名、工程名上得不任何关联，本身也并不特殊，那在日后或是其它开发者在观看源代码时会造成相当的困难。

所以我们推荐使用 const 或是#define 来给这个数字命个名，这样在查看代码时就会轻松许多了！

类成员函数命名方式、PME 构想

PME 构想是当今最为流行的软件工程模型，《古月》在设计时就使用了 PME 构想。

PME 是属性/特性、方法、事件三个单词的首字母缩写：
Property、Method、Event。

属性，一般情况下的表现形态为变量。
方法，一般情况下的表现形态为函数，使用和普通函数一样的命名方式。
主动事件，一般情况下的表现形态为回呼函数，全部以 On 为前缀。
被动事件，一般情况下表现形态为虚拟函数，全部以 Proc 为前缀。

主动事件是指这个对象可能会引起的事件，这些事件一般由开发者指定处理函数；
被动事件是指这个对象可能要处理的事件，这些事件函数就是在对象中已经定义的函数，将由对象的上一级来调用。

PME 类示例

```
class NPC
{
public: //属性

    __property wchar_t *Name={read=name,write=SetName};        //NPC 名称
    __property int HP={read=hp};                               //NPC 的生命值

public: //事件

    //当玩家对当前 NPC 说话时会触发的事件
    void (__closure *OnPlayerChat)(Player *man,wchar_t *str);

    //当玩家攻击当前 NPC 时会触发的事件
    void (__closure *OnPlayerAttck)(Player *man,Attack *data);

public: //方法

    bool GoTo(GameMap *map,int x,int y);                       //命令 NPC 到指定地点
    bool Attack(Player *man);                                  //命令 NPC 攻击指定玩家
};
```

排版

1.代码缩进空格默认为 4 个。

2.在相对独立的代码段之间请加入空行，比如以下代码：

```
int result=check();
if(!result)
{
    //.....
}
return(result);
```

应书写如下：

```
int result=check();

if(!result)
{
    //.....
}

return(result);
```

3.尽量每个语句写一行，如下面的代码：

```
rect.width=0;rect.height;
```

应书写如下：

```
rect.width=0;
rect.height=0;
```

4.if、for、do、while、case、switch、default 等语句要独占一行，花括句要独占一行。
如下面的代码：

```
for(int i=0;i<10;i++){
if(i==index)return(i);}
```

应书写如下：

```
for(int i=0;i<10;i++)
{
   if(i==index)return(i);
}
```

5.程序的代码段应采用缩进风格，与上一级的代码段多一级。花括号做为母段的内容，不参与缩进。如下面的代码：

```
switch(result)
{
case 0:..
case 1:
}

if(...)
    {
        //...
    }
```

应书写如下：

```
switch(result)
{
    case 0:..
    case 1:
}

if(...)
{
    //...
}
```

附一：Borland 扩展关键字__closure

(摘自《C++Builder 开发人员指南》)

__closure 关键被用来声明一个特殊类型的指针指向成员函数。不同于一般的C++成员函数指针，闭合包含对象指针。

在标准的 C++中，可把一个派生类的实例分配给基类指针；然而，不能把一个派生类的成员函数分配给基类成员函数指针。

下列代码说明这种情况：

```
class derived:public base
{
  public:
    void new_func(int i);
};

void (base::*bptr)(int);
bptr=&derived::new_func; //不合规定的用法
```

然而，__closure 扩展允许在 Borland C++中这么做。

```
class MyObject
{
  double MemFunc(int);
};

double CallFunc(MyObject *obj)
{
  double (__closure *bptr)(int);

  bptr=obj->MemFunc;

  return(bptr(0));            //等同于 return(obj->MemFunc(0));
}
```

附二：Borland 扩展关键字__property

(摘自《C++Builder 开发人员指南》)

__property 关键字在类声明中声明属性。属性只能在类中被声明。对于属性数组，数组的索引可以是任何类型。

使用__property 关键字可以指定一个属性的读或写方法。例如：

```
class MyObject
{
   int mp,st;

   int GetHP();
   void SetHP(int);
   void SetMP(int);

public:

   __property int HP={read=GetHP,write=SetHP};
   __property int MP={read=mp,write=SetMP};
   __property int ST={write=st};
};
```

附三：Microsoft 扩展关键字 property

微软的 property 关键字与 Borland 的__property 关键字作用类似，下面的代码取自 Microsoft Windows SDK。

```
// declspec_property.cpp
struct S
{
   int i;
   void putprop(int j)
   {
      i = j;
   }

   int getprop()
   {
      return i;
   }

   __declspec(property(get = getprop, put = putprop)) int the_prop;
};

int main()
{
   S s;

   s.the_prop = 5;

   return s.the_prop;
}
```

谢谢!

Thank you for Look!