



# Game Development Seminar

group 01: coding style



# coding style – why?

- uniform appearance
- clean
- easy to read
- easy to maintain



## coding style - names

- make **names that fit**, if you find all your names could be *Thing* and *Dolt* then you should probably revisit your design
- **Suffixes** are sometimes helpful. For example, if your system uses **agents** then naming something *DownloadAgent* conveys real information



# coding style - names

- **Prefixes** are sometimes useful:
- */s* - to ask a question about something.  
Whenever someone sees */s* they will know it's a question
- *Get* - get a value
- *Set* - set a value



## coding style - names

- include **units** in names, for example use variables like `objectDistancekm` or `symbolWidthPixels`
- don't use upper case abbreviations: instead of calling a procedure `getODBCSource` change it to `getOdbcSource` as the “ODBCS” is confused



# coding style - names

- Use **upper case** letters as word separators, **lower case** for the rest of a word
- **First character** in a name is upper case
- No underbars ('\_')
- You may use prefix 'C' to distinguish between classes and structures (class **CSound3D**)



# coding style - names

## ■ **Class Attribute Names**

- Attribute names should be prepended with the character 'm'.
- After the 'm' use the same rules as for class names.
- 'm' always precedes other name modifiers like 'p' for pointer or 'r' for references



# coding style - names

- **Method Argument Names**

- The **first character** should be lower case

- All word beginnings after the first letter should be upper case as with class names

- example:

```
int StartYourEngines( Engine&  
rSomeEngine, Engine& rAnotherEngine);
```





# coding style - names

- **Variable Names on the Stack**
- **do not** use all lower case letters
- **do not** use '\_' as the word separator
- reason: either it's hard to read or hard to type



# coding style - names

## ■ **Pointer Variables**

- pointers should be **prepended by a 'p'** in most cases
- place the \* **close to the pointer type** not the variable name
- `String*` pName= new String;  
`String*` pName, name, address; *// note, only pName is a pointer*



# coding style - names

- **Global Constants:**

- Global constants should be all caps with '\_' separators

- `const int A_GLOBAL_CONSTANT= 5;`

- **#define and Macro Names:**

- Put #defines and macros in all upper using '\_' separators



# coding style - names

- **Enum Names:**

- Labels All Upper Case with '\_' Word Separators

- **Example**

- ```
enum PinStateType {  
    PIN_OFF,  
    PIN_ON };
```



## coding style - names

- Make a Label for an Error State

- **Example:**

```
enum { STATE_ERR, STATE_OPEN,  
STATE_RUNNING, STATE_DYING};
```



# coding style - braces

- **Braces { } Policy**
- Of the three major brace placement strategies two are acceptable, with the first one listed being preferable:
- Place brace under and inline with keywords



# coding style - braces

- **Examples**

- if (condition)

- {

- ...

- };

- while (condition)

- {

- ...



## coding style - braces

- If you use an editor (such as vi) that supports brace matching, this is a much better style. Why? Let's say you have a large block of code and want to know where the block ends. You move to the first brace hit a key and the editor finds the matching brace.





# coding style - braces

- **Example:**

- ```
if (very_long_condition &&  
    second_very_long_condition)  
{  
    ...  
}  
else if (...)  
{  
    ...
```



# coding style - braces

- **One Line Form**

- `if (1 == somevalue)`  
    `somevalue = 2;`

- **Justification:**


It provides safety when adding new lines while maintainng a **compact readable form**.



# coding style - format


- **Consider Screen Size Limits**

- Some people like blocks to fit within a common screen size so scrolling is not necessary when reading code
  - **restriction to 80 characters/line**  
(Even though with big monitors we stretch windows wide our printers can only print so wide. And we still need to print code.)



## coding style - comments


- Every **parameter** should be **documented**. Every **return code** should be documented. All **exceptions** should be documented. Use **complete sentences** when describing attributes. Make sure to think about what other resources developers may need and encode them in with the @see attributes.



# coding style - comments

- **Example:**

- ```
/**  
 * Assignment operator.  
 *  
 * @param val The value to assign to this object.  
 *  
 * @return A reference to this object.  
 */  
XX& operator=(XX& val);
```



# coding style - comments

## ■ Block Comments


- Use comments on starting and ending a Block:

```
{  
    // Block1 (meaningful comment about Block1)  
    ... some code  
    {  
        // Block2 (meaningful comment)  
        ... some code  
    } // End Block2  
} // End Block1
```



# coding style – static variables

- **Make Functions Reentrant**
- Functions should not keep static variables that prevent a function from being reentrant.
- Functions can declare variables static. Problems happen when the function is called one or more times at the same time. This can happen when multiple tasks are used.



# coding style – header files

- **Use Header File Guards**
- Include files should protect against multiple inclusion through the use of macros that "guard" the files.
- `#ifndef filename_h`  
`#define filename_h`  
`...`  
`#endif`





## coding style – *switch* Formatting

- **Falling** through a case statement into the next case statement **shall be permitted** as long as a comment is included.
- The ***default*** case should always be present and trigger an error if it should not be reached, yet is reached.
- If you need to create variables put all the code in a **block**.



# coding style – *switch* Formatting

## ■ Example:

```
switch (...)
{
case 1:
    ...
    // FALL THROUGH
```

```
case 2:
    {
        int v;
        ...
    }
    break;

default:
}
```



## coding style – declaration blocks

- Block of **declarations should be aligned**, this preserves clarity.
- Similarly blocks of initialization of variables should be **tabulated**.
- The ‘&’ and ‘\*’ tokens should be adjacent to the **type**, not the name.

# coding style – declaration blocks

## ■ Example:

- `DWORD mDword;`  
`DWORD* mpDword;`  
`char* mpChar;`  
`char mChar;`

```
mDword = 0;  
mpDword = NULL;  
mpChar = NULL;  
mChar = 0;
```

aligned and tabulated



## coding style – Macros

- Replace Macros with Inline Functions.  
In C++ macros are not needed for code efficiency. **Use inlines.**
- **Be Careful of Side Effects**  
Macros should be used with caution because of the potential for error when invoked with an expression that has side effects.



# coding style – Macros

## ■ Example:

```
#define MAX(x,y) (((x) > (y) ? (x) : (y)) // get maximum
```

## ■ inline int max(int x, int y)

```
{  
    return (x > y ? x : y);  
}
```

## ■ Side Effects:

```
MAX(f(x),z++);
```




## coding style – Macros

- If you use macros: *Always Wrap the Expression and the Parameters in Parenthesis*
- `#define ADD(x,y) x + y`

must be written as


```
#define ADD(x,y) ((x) + (y))
```



# coding style – Initialization

- You shall **always** initialize variables.  
*Always. Every time.*
- **Justification:**  
More **problems** than you can believe are eventually traced back to a pointer or variable left **uninitialized**. C++ tends to encourage this by spreading initialization to each constructor.





# coding style – Initialization

## ■ **Initializing Objects:**


Objects with multiple constructors and/or multiple attributes should define a private **Init()** method to initialize all attributes. If the number of different member variables is small then this idiom may not be a big win and C++'s constructor initialization syntax can/should be used.



# coding style – Initialization

```
class Test
{
    public:
    Test()
    {
        Init(); // Call to common object initializer
    }

    Test(int val)
    {
        Init(); // Call to common object initializer
        mVal= val;
    }
}
```



# coding style – Alignment

- There should be only **one statement per line** unless the statements are very closely related.

The reasons are:

- The code is **easier to read**.
- **Documentation** can be added for the variable on the line.
- It's clear that the variables are **initialized**.



## coding style – Code Reusage

- **Ask! Email a Broadcast Request to the Group**

This simple technique is rarely done. For some reason programmers feel it makes them **seem** less capable if they ask others for help. This is **silly**! If you need a piece of code, email to the group asking if someone has already done it. The results can be surprising.



## coding style – Code Reusage

- **Tell! When You do Something Tell Everyone.**


Let other people know if you have done something reusable. Don't be shy. And don't hide your work to protect your pride. Once people get in the habit of sharing work everyone gets better.



## coding style – Code Reusage

- **Don't be Afraid of Small Libraries**

One common enemy of reuse is people not making libraries out of their code. A **reusable class may be hiding in a program directory** and will never have the thrill of being shared because the programmer won't factor the class or classes into a library.



# coding style – Comments

- **Make Gotchas Explicit**

- **:TODO:** topic


Means there's more to do here, don't forget.

- **:BUG: [bugid] topic**

means there's a Known bug here, explain it and optionally give a bug ID.

- **:KLUDGE:**

When you've done something ugly say so and explain how you would do it differently next time if you had more time.



# coding style – Comments

- **:TRICKY:**

Tells somebody that the following code is very tricky so don't go changing it without thinking.


- **:WARNING:**

Beware of something.

- **:COMPILER:**

Sometimes you need to work around a compiler problem. Document it. The problem may go away eventually.





# coding style – Comments

## ■ Example:

- `// :TODO: tmh 960810: possible performance problem`  
`// We should really use a hash table here but for now`  
`// we'll use a linear search.`  
`//`  
`// :KLUDGE: tmh 960810: possible unsafe type cast`  
`// We need a cast here to recover the derived type. It`  
`// should probably use a virtual method or template.`



# coding style – Code Reusage

## ■ **Directory Documentation:**

Every directory should have a README file that covers:

- the purpose of the directory and what it contains
- a one line comment on each file. A comment can usually be extracted from the NAME attribute of the file header.



# coding style – Code Reusage

- cover build and install directions
- direct people to related resources:
  - ☐ directories of source
  - ☐ online documentation
  - ☐ paper documentation
  - ☐ design documentation
- anything else that might help someone



## coding style – Law of Demeter

- The *Law of Demeter* states that you **shouldn't access a contained object directly** from the containing object, you should use a method of the containing object that does what you want and accesses any of its objects as needed.
- The purpose of this law is to **break dependencies** so implementations can change without breaking code.




# coding style – Commenting Out

- Sometimes large blocks of code need to be commented out for testing.
- The easiest way to do this is with an **#if 0** block:

```
#if 0
```

```
    lots of code /* comments */
```


```
#endif
```



# coding style – Accessors


## ■ Get/Set

```
class X
{
public:
    int    GetAge() const    { return mAge; }
    void    SetAge(int age)    { mAge= age; }
private:
    int mAge;
}
```



## coding style – Accessors

- The **problem with Get/Set** is twofold:
- It's **ugly**. Get and Set are strewn throughout the code cluttering it up.
- It **doesn't treat attributes as objects** in their own right. An object will have an assignment operator. Why shouldn't age be an object and have its own assignment operator?



# coding style – Accessors


## ■ Attributes as Objects:

```
■ class X
{
public:
    int      Age() const    { return mAge; }    //⇔ get
    int&      rAge()        { return mAge; }    //⇔ set

    const String&  Name() const { return mName; }
    String&        rName()      { return mName; }

private:
    int      mAge;
    String   mName;
}
```





## coding style – Accessors

- This approach is also more **consistent with the object philosophy**: the object should do it. An object's = operator can do all the checks for the assignment and it's done once in one place, in the object, where it belongs. It's also clean from a name perspective. **When possible use this approach to attribute access.**



# coding style – File Extensions

- **In short: Use the *.h* extension for header files and *.cpp* for source files.**
- **Why?**

The short answer is as long as everyone on our project agrees it doesn't really matter as the build environment should be able to invoke the right compiler for any extension.



# coding style – Magic Numbers

- A magic number is a bare naked number used in source code. It's magic because no-one has a clue what it means including the author inside 3 months.

**Don't use them!**



# coding style – Magic Numbers

## ■ Example:

```
■ if      (22 == foo)    { start_thermo_nuclear_war(); }  
  else if (19 == foo)    { refund_lotso_money(); }  
  else if (16 == foo)    { infinite_loop(); }  
  else                  { cry_cause_im_lost(); }
```

## ■ Alternatives:

```
■ #define  PRESIDENT_WENT_CRAZY (22)  
  const int WE_GOOFED= 19;  
  enum  
  { THEY_DIDNT_PAY= 16 };
```



# coding style – Acknowledgement

- **The Original author:**

Todd Hoff

The C++ Coding Standard

- **Last Modified: 2005-02-24**

- <http://www.possibility.com/Cpp/CppCodingStandard.html>

- This presentation is *not* an exact copy, there are slight changes everywhere.