Plugin Code Style Guide

Introduction

The C++ language has many powerful features, but this power brings with it complexity, which in turn can make code more error-prone and harder to read and maintain. The goal of this guide is to help you manage this complexity by describing in detail the major dos and don'ts of writing C++ code for plugins. We recommend you enforce consistency to keep your code base manageable. Creating common idioms and patterns makes code much easier to understand. In some cases there might be good arguments for changing certain style rules, but nonetheless it is best to preserve consistency as much as possible. A major benefit is it will allow us to better understand your code snippets when asking for help on the Plugin Cafe forum or via direct support.

Another issue this guide addresses is that of C++ feature bloat. C++ is a huge language with many advanced features. In some cases we constrain, or even ban, the use of certain features. We do this to keep code simple and to avoid the various common errors and problems that these features can cause. This guide lists these features and explains why their use is restricted.

---

1. Basic C++ Language

1.1 C++ 11

The following C++ 11 features are currently supported:

- Move semantics / R-Value references
- `std::is_pod`, `std::is_scalar`, `std::is_base_of`, `std::is_function`, `std::is_polymorphic`, `std::is_convertible`
- `static_assert`
- `auto`
- `decltype`
- Range-based for loops
- Extern template
- Lambda expressions and closures
- Explicit overrides and final
- Right angle brackets
- `nullptr`
- Thread-local storage
- Strongly typed enums
- Brace-or-equal initializers (in-class member initializers)

Note: Any other C++ 11 features must not be used.

1.2 For Increments

Do not use for statements that use floating point values as an increment. Float increments can lead to infinite loops or unexpected behavior.

```
Float maxRange, minRange, increment;
...
if (increment == 0.0) return false;

Int cnt = SAFEINT((maxRange – minRange) / increment);

for (Int i = 0; i < cnt; i++)
  currentPosition = minRange + (maxRange - minRange) * Float(i) / Float(cnt);
  ...

… initialize and free at some point…
```

1.3 ANSI C Routines

Don’t use ANSI C routines in your code if there is a library replacement. E.g. `memset`, `memcpy`, `printf`, `scanf` etc. should never be touched.

1.4 Casts

Even though they look a little messy, use `static_cast` and `const_cast` for all non-arithmetic datatypes. This has several advantages: casts can be searched easier and they also are much safer as you cannot cast into a completely different type accidentally.

```
Float b = (Float)a; // arithmetic datatype

const MyObject* constObj = static_cast<const MyObject*>(inputObject);
MyObject* obj = const_cast<MyObject*>(constObj);
```

1.5 Goto

The `goto` statement should only be used if it is absolutely the best choice for an algorithm's implementation. Generally, your code should be structured so that it can return at any point and that it makes sure everything is cleaned up properly.

---

2. CINEMA 4D C++ specifics

2.1 External Libraries

Do not use external libraries (e.g STL or BOOST) unless absolutely necessary.

2.2 Threading

Be aware of threading!

- GUI interaction must never happen in a thread. Instead, offload jobs to the main thread queue.
- Input states (e.g. from the keyboard or mouse) must never be queried in a thread.
- Do not write to global structures from a thread.
- Any shared resources must be properly protected by SpinLocks or Semaphores.
- Be aware of how deadlocks are created.

2.3 Global Static Classes

Static or global variables of class type (e.g. using something like `String g_mySharedStringVariable`) are forbidden due to the indeterminate order of construction and destruction. The same applies to `AutoMem`.

2.4 Array Classes

Only use the classes **BaseArray**, **BlockArray**, **PointerArray** or derived classes. Whenever you access arrays make sure that any access is within boundaries, otherwise you'll provoke a crash and undefined behaviour. You can only omit those checks for speed critical areas. In that case, you should document in the routine's header that a variable needs to be within a certain range.

```
Int MyRoutine(Float x, BaseArray<Int>& arr) // pass an array with 100 elements
{
  Int index = Int(x * 100.0);
  if ((UInt)index >= 100) // by casting to UInt you can merge two checks into one as any negative value
         casted to UInt exceeds 100
    return -1; // return error code

  return arr[index];
}
```

2.5 Range-Based for Loops and Iterators

When iterating, use references (unless specifically needed otherwise) so that no unnecessary data will be copied. Always use `const` if no modification is necessary. If it doesn't compile, fix your classes so they properly support constness. If you need additional functionality obtain an auto-iterator with `ForEach::Get()` or an erase-iterator with `ForEach::EraseIterator`.

```
BaseArray<Vector> arr;
...
for (Vector& value : arr)
{
  value = Vector(1.0);
}

for (const Vector& value : arr)
{
  DiagnosticOutput("@", value);
}

// EraseIterator allows you to still use range-based for loops
for (auto& it : ForEach::EraseIterator(arr))
{
  if (*it == valueToErase)
    it.Erase();
}
```

2.6 Magic Numbers

Use constants instead of magic numbers wherever possible. The meaning of arbitrary numbers is often not clear to the reader. An explicit name helps readability.

```
const Int DAYS_IN_WEEK = 7;
const Int DAY_MONDAY = 0;
const Int DAY_SUNDAY = 6;
Int day;
...
if ((day % DAYS_IN_WEEK) == DAY_SUNDAY)
  CallJob();
```

---

3. Error Handling

3.1 Defensive Coding

Try adopting a defensive style - add additional checks, even if you are sure these conditions do not happen. At a later time, circumstances may change, making those particular conditions come true. Make sure your source code compiles free of warnings on all platforms and targets.

3.2 Exceptions & RTTI

C++ exceptions and RTTI will not be used in any of our projects unless external libraries require it (also RTTI operators typeid and dynamic_cast cannot be used).

3.3 Memory Checks

Every memory allocation must be checked. It is not sufficient to check and leave your scope though. You must make sure that a class is not in an uninitialized state and will still properly function according to specifications. It is highly recommended that you implement memory checks during development and not at the very end of a development cycle. Instead of using memory blocks always use arrays (e.g. **BaseArray**, **BlockArray**, **PointerArray**). They're much more flexible, safe (you get additional safety checks in debug builds), reduce code size and offer the same performance.

```
class Test
{
public:
  Test() { }

  Bool Init()
  {
    return arr.Resize(100);
  }

  Int GetCount() { return arr.GetCount(); }

private:
  BaseArray<Char> arr;
}
```

3.4 Memory Allocations

Any class allocation (except for system or library specific code) must be done using ::NewObj() and ::DeleteObj(). Any memory allocation must be done using ::NewMem() and ::DeleteMem(). Memory obtained with ::NewObj() and ::NewMem() is NOT initialized, so you must make sure that the class/struct constructors properly initialize all member variables. To obtain cleared memory, use ::NewMemClear(). For constructors, this is not available. As a side note, AutoMem is a convenient way to free memory when your scope is left in case of an error.

Circumventing 's memory management and using the standard libs allocators will result in lower speed and will keep you from utilizing 's leak detection utilities.

3.5 Divisions by zero

Any floating point division must be checked for 0.0 unless the algorithm specifically doesn't require this check. An example where it is not necessary is e.g. `x /= (1.0 + vectorA * vectorA)` where the divisor never can become 0.0. Otherwise, a test must be done. It cannot be stated often enough that floating point calculations are imprecise... (`c - d` below) can be different to 0.0 and then in the next code line be 0.0. This is not even uncommon with optimizing compilers.

Any integer division must also be checked for 0, however the strict requirements aforementioned don't apply here.

```
Float x, tmp = c - d;
if (tmp != 0.0)
  x = (a - b) / tmp;
else
  x = ...;
```

3.6 Type Conversions

Always be careful with type conversions as they are a frequent cause for crashes!
E.g. if you want to calculate an index within a 2-dimensional array. Let's assume both dimensions are larger than 32767 units. In that case the index needs to be of type ::Int (64-bit), otherwise you'd get an overflow. What is often overseen though is that it is not enough to just change the index to ::Int:

```
Int32 xRes = LIMIT<Int32>::MAX, y = LIMIT<Int32>::MAX, x = 0;
Int index = xRes * y + x;
```

If you take a look in the debugger you'll see the following:

```
xRes    2147483647
y       2147483647
index 1
```

The calculation is done with ::Int32 and then at the end the result is casted to ::Int (when it is too late).

So the only correct way to write this (in case you want to keep `x`, `y` and `xRes` as ::Int32) is:

```
index = (Int)xRes * y + x;
```

or

```
index = xRes * (Int)y + x;
```

but not

```
index = xRes * y + (Int)x;
```

Another frequent conversion problem is if you go from ::Float to ::Int. In many places a 3D point is projected onto the screen, converted to Int and then clipped against the screen boundaries. This will always fail!

Let's take a look why:

```
Float n = 1.0e10;
Int m = (Int)n;
```

In the debugger you will see:

```
n           1.0000000000000000e+020
m           -9223372036854775808
```

So if you want to perform a proper clipping you have two possibilities:

a) Clip floating point values first (on a float basis) and then convert to Integer

```
Float xClamped = Clamp(xInput, (Float)left, (Float)right);
Int x = (Int)xClamped;
```

b) Use **SAFEINT**(32) which makes sure that if a floating-point value exceeds the minimum/maximum integer number then the minimum/maximum will be set

```
Int x = SAFEINT(xInput);
x = Clamp(x, left, right);
```

3.7 Asserts, Stops and Output

For Asserts and Stops there is DebugAssert() / DebugStop() as well as CriticalAssert() / CriticalStop().

Generally, any of those functions should only be used if there is (or to check for) a severe problem in your code. E.g. a DebugAssert() could check if the sum of segments matches the total count, or if a pointer is != `nullptr`, as anything else shouldn't (or can't) be the case. A DebugAssert() shall not be set for any conditions that are relevant only to you (e.g. stop if the name of an object is 'default') or are not critical for the algorithm (e.g. stop if a file handler can't open a file).

DebugAssert() / DebugStop() are like their Critical counterparts, but will be removed during release builds for speed reasons. If the code is speed uncritical try to utilize the Critical versions.

When outputting a value you can choose from DiagnosticOutput(), WarningOutput() and CriticalOutput(). CriticalOutput() and WarningOutput() will by default be visible to the end-user, DiagnosticOutput() needs to be manually enabled. CriticalOutput() additionally enforces a CriticalStop() compared to WarningOutput(). CriticalOutput() and WarningOutput() print header information about source code file, line and current time.

---

4. File Structure

4.1 File Names

To avoid trouble on case sensitive file systems or when compiling cross-platform, the names of any source files are lowercase and cannot contain any white spaces or special characters (ASCII characters >=128).

4.2 Header Files

Every *.cpp file should have an associated *.h file (unless it is e.g. a main.cpp)

4.3 Header Guards

All header files must be protected by define guards containing the file name in upper cases with the dot replaced by an underscore and two underscores appended.

```
#ifndef MYHEADERFILE_H__
#define MYHEADERFILE_H__

#include “c4d.h”

void MyFunction();

#endif // MYHEADERFILE_H__
```

4.4 File Length

Keep the number of code lines within a file within reason. Long files might be better grouped into smaller ones and improve readability. If your file exceeds 5000 lines it should definitely be split into parts.

4.5 Includes

Only include header files when necessary, use forward declarations of classes if possible.

---

5. Datatypes

5.1 Basic Datatypes

Always use C4D's datatypes ::Int, ::Int32, ::Int64, ::Float, ::Float32, ::Float64, ::Bool, ::Char, etc. unless system specific code requires otherwise. You should use `nullptr` instead of `NULL` or `0`.

5.2 Boolean Values

Use `true` and `false` for boolean variables (not `TRUE` or `FALSE`, etc.). This will ease the code conversion to C++11 booleans.

5.3 Size Calculations

Only use `size_t` when it is explicitly necessary. Otherwise, ::Int is the better choice, as it is signed and allows for subtractions and comparisons. Use the macro SIZEOF instead of `sizeof` to automatically cast to the datatype ::Int.

Note: ::Int is 32-bit on 32-bit systems, so a multiplication can result in an overflow (use ::Int64 instead).

5.4 Unsigned Datatypes

Try to use the unsigned datatypes ::UInt, ::UInt32 and ::UInt64 only if there is a real benefit vs. ::Int, ::Int32 and ::Int64.

5.5 Indices

Any index or size should be of type ::Int. Do expect that memory can become bigger than 2 GB. If you need to be able to specifically handle more than 2GB data on 32 bit systems (e.g. for File I/O) you must use ::Int64 instead of ::Int, as ::Int is defined as 32-bit on a 32-bit system and 64-bit on a 64-bit system.

---

6. Comments

6.1 Documentation Requirement

Code must be commented. It makes it easier for other developers to understand what your code does, and why, thus saving lots of time later. When writing code comments, try to describe why things are done, not what is done. It is easy to find out what is done by just reading the code. But it's hard to find out the reason why things are done. Therefore, always describe the purpose of the code in your comments.

Comments that just repeat the code below are counterproductive. Even worse, often the comment will be outdated and not reflect what is happening in the code anymore. For interfaces describe valid ranges for variables, return values, special conditions and anything that is non-obvious (e.g. if a function class has a certain performance or threading behaviour, if it interacts with another portion of the code or if it needs to be called within a certain context). If functions or members have to be called in a certain order this needs to be documented (unless it is obvious e.g. in the case of Init()).
Every class definition should have an accompanying comment that describes what it is for and how it should be used.

6.2 Unnecessary Comments

Try to make your source more readable instead of adding lots of comments. Comments do not make up for bad code. A good way to add readability is to introduce constants and assign inbetween calculations to clearly named variables.

Here there is no need for any comments. The code is clear and modifications to the code don’t require modifications to the comments as well.

```
const Int deleteObjectThreshold = 1024;
Bool noMoreSegmentsAvailable = 2 * seg > maxSegments;

if ((value >= deleteObjectThreshold) || noMoreSegmentsAvailable)
  DeleteObject(x);
```

---

7. Formatting

7.1 Tab Width & Use

We use use a tab width of 2. Furthermore, tabs are only used at the beginning of a line and to visually align elements. Tabs should not be used after an opening parenthesis or as a replacement for a single space.

7.2 Spaces

Use spaces like in your natural language. Spaces should be set after a semicolon or comma and after comparisons or assignment operators. Arithmetic calculations should be separated by a space.

```
for (Int x = 0; x < 10; x++)
if (x == y)
x = (a * b) / (c * d)
```

7.3 Compact Algorithms

Prefer small and focused functions and member routines, as this makes it easier for others to get a general idea of the algorithm and allows the compiler to create more efficient code. As a guideline routines should not get longer than two full screen pages – otherwise think how it can be split up into smaller tasks.

7.4 Bracket Alignment

Opening/Closing brackets should be aligned at the same X position or be within the same code line. However, be aware that placing the code in the same line makes debugging harder as you can't set a breakpoint for the execution part and won't see if the condition was met or not.

```
if (x == 0.0)
{
  ...
}

class Test
{
  Int GetValue() { ... };
}
```

7.5 Use of Brackets

Don't introduce additional brackets if there is no need for it and if it doesn't improve readability.

```
x = a > 0 ? (2 * a) : b;

if ((z == a) ^ (y == a) == b)
if ((x == 5.0 && y == 2.0) ||
    (x == 2.5 && y == 1.0))
```

7.6 Switch Statements

Switch statements are formatted in the following way:

- The `case` statements are indented by one tab compared to the `switch` statement.
- The opening/closing brackets are at the same horizontal position as the `case` statements.
- The `break` statement is indented by one tab compared to the `case` statements and is written before the closing bracket.
- If no opening/closing brackets are used both `case` and `break` can go on the same line (but this should only be used if all cases follow the same formatting).
- Each `case` must stop with either `MAXON_SWITCH_FALLTHROUGH`, `break`, `continue` or `return`.

```
switch (i)
{
  case 0:
  {
    break;
  }

  case 1:
    break;

  case 2: return true;
}
```

7.7 Switch Statements Without Break

If you specifically design a fallthrough for a switch statement `MAXON_SWITCH_FALLTHROUGH` must be used to document the code. Note that this only needs to be done if the fallthrough happens after some code was written, not for a list of multiple cases.

7.8 If - Statements

Don't use assignments in if statements.

Always start a new line after the if statement, as this makes debugging easier.

```
a = GetFirstNode();
if (a == b)
  return false;
```

7.9 Else Statements

- If there is an `else` statement it needs to be followed by a new line.
- Nested `if` / `else` statements must use brackets.
- If an `if` – block has brackets, the `else` block must have brackets too and vice versa.
- After `if` / `else` there always must be a new line.

```
if (x)
  do A
else
  do B

if (x)
{
  do A;
}
else
{
  if (y)
    do B;
  else
    do C;
}
```

7.10 Spaces and pointers

Do not put spaces around the period or arrow operators for pointers and classes. Pointer operators have no space after the * or the &. When declaring a pointer variable or argument, group the asterisk to the type, unless you are declaring many variables of the same type at once.

```
Int* l;
const String& str;
String*& str;

// but allowed for better readability:
Int *a, *b, *c, *d;
```

7.11 Ifdefs

`#if`, `#ifdef`, `#elif`, `#else`, `#endif` statements should be aligned at horizontal position 0. If they are nested, indentation should be used. `#undef` should never be used unless absolutely necessary, and never goes into a header file.

```
class MyTest
{
  ...
public:
  Int GetIndex() const
  {
#ifdef __PC
    return 0;
#else
    return 1;
#endif
  }
};

#ifdef __PC
  #if _MSC_VER >= 1600
    #define _HAS_NULLPTR
  #else
    #undef _HAS_NULLPTR
  #endif
#else
  #undef _HAS_NULLPTR
#endif
```

7.12 Operators

Operator names need to be treated like regular function names.

```
Bool operator ==(const AutoIterator& b) const;
```

7.13 Do-While Loops

Use exactly this formatting:

```
do
{
  ...
} while (condition);
```

7.14 For - Statements

- Always start a new line after the for statement.
- Use brackets if the for contains more than one line of code.

```
for (Int x = 0; x < 10; x++)
{
  if (x == 0)
    sum += 5;
  else
    sum += x;
}
```

---

8. Naming

8.1 Enums

Use strongly typed enum constants instead of defines wherever possible. This vastly enhances source code readability. The convention is to type the enum in capital letters. List elements do not repeat the enum's name.

```
enum class MOUSEDRAGRESULT
{
  ESCAPE    = 1,
  FINISHED  = 2,
  CONTINUE  = 3
} ENUM_END_LIST(MOUSEDRAGRESULT);
```

8.2 Flag Sets

Flags (unless they're bit arrays) must be defined as strongly typed enums. If you need a default element that equals the value 0 you should define the list element NONE. To test a flag set for a bit use `CheckFlag(value, bit)`, e.g. `CheckValue(myFlags, MOUSEDRAGFLAG::NOMOVE)`. Do not cast enums to integers for such tests.

```
enum class MOUSEDRAGFLAGS
{
  NONE                  = 0,
  DONTHIDEMOUSE         = (1 << 0),
  NOMOVE                = (1 << 1),
  AIRBRUSH              = (1 << 4)
} ENUM_END_FLAGS(MOUSEDRAGFLAGS);
```

8.3 Constants, Macros and Enums

Constants, Macros and enumerations only use uppercase letters. Words can, but don't have to be separated by an underscore. Use `const` for constants instead of `define`. If you use function macros, try to add the semicolon at the end so that it looks like a function call.

```
const Float THIRD = 0.333333333;

WarningOutput("Test");
```

8.4 Variable names

- Variable names start with a lower case and use an upper case for subsequent Words.
- Hungarian notation is not recommended.
- Give descriptive names (e.g. 'numErrors' instead of 'nerr'). Do not use abbreviations unless it is known clearly outside of your code ('rgbBuffer' is ok, but 'iLock' for internal lock is not sufficient).
- The names 'id' and 'auto' cannot be used, as they conflict with different operating systems or compilers.
- Try to get away from names like 'i', 'j', 'k', 'a', 'b', 'c' unless you're using a trivial loop or situation that only uses a few lines of code.
- Try to name your variables according to their true meaning – e.g. 'arrayIndex' instead of 'i', 'sizeInKB' instead of 'ksize', 'segmentCount' instead of 'scnt' etc. The longer a while or for loop is (especially when it doesn't fit on the screen anymore) the clearer a variable must be named. If your code is nested in three loops and the counters are all named 'i', 'j' and 'k', it will become extremely difficult to read.

```
Int stringTable;

for (Int polygonIndex = 0; polygonIndex < polygonCount; polygonIndex++)
{
  for (Int side = 0; side < maxSides; side++)
  {
    marked[polygon[polygonIndex][side]] = true;
  }
}
```

8.5 Member Variables

Member variables of classes, but not structs, should always be private or protected and get an underscore at the beginning. See also **Structs**.

```
class MyTest
{
private:
  Int _a;
}
```

8.6 Type Names

Type names (class members, functions and classes) start with a capital letter for each new word, with no underscores.

```
class MyNewClass;

Bool IsEqual();
```

8.7 Global Variables

Global variables start with g_ and then continue with the standard naming conventions.

```
Int g_var;
Int g_macAddress;
```

8.8 Templates

Template Parameters only use uppercase letters. Words don't have to be separated by an underscore. Use `typename` instead of `class` for template parameters. After the word `template` there is always a space.

```
template <typename SORTCLASS, typename ITERATOR> class BaseSort;
```

8.9 Pairing

For members or variables that do the opposite thing use the correct name pairing, e.g.

- Get and Set
- Add and Remove
- Create and Destroy
- Start and Stop
- Insert and Delete
- Increment and Decrement
- Old and New
- Begin and End
- First and Last
- Up and Down
- Min and Max
- Next and Previous
- Open and Close
- Show and Hide
- Suspend and Resume

---

9. Classes and Parameters

9.1 Class Layout

Use the following order of declarations within a class: public before protected, protected before private, and methods before data members (variables). The keywords `public`, `protected` and `private` are aligned with the keyword `class`.

```
class MyTest
{
public:
  MyTest();
  ~MyTest();
protected:
  Int GetIndex();
private:
  void Free();

protected:
  Int _a,_b,_c;
private:
  Int _d;
};
```

9.2 Class Accessors

Class accessors should be named using the words ‘Get’ and ‘Set'.

```
class Example
{
  public:
    Int  GetCount();
    void SetCount(Int count);

  private:
    Int _count;
}
```

9.3 Class Initialization

Class constructors should only set member variables to their initial values. Any complex initializations should go in an explicit Init() method. No memory must be allocated in the constructor, unless there is a specific reason for it. In general, we recommend brace-or-equal initializers for initialization (unless the initialization is performance-critical) which have the added benefit that you don't have to write repeat code for multiple constructors. If you use member-initializer lists, don't write each member on its own line. Instead leave them on one line, or split it into two lines with one tab indentation.

```
class Test()
{
public:
  Test()
  {
    _mem = nullptr;
  }

  Bool Init()
  {
    _mem = NewMem(Char, 100);
    return _mem != nullptr;
  }

private:
  void* _mem;
};

class Test()
{
public:
  Test() : _val(5) { } // member-initializer list

private:
  Int _val;
};

class Test()
{
public:
  Test() : _val1(5), _val2(6), _val3(7), _val4(8), _val5(9), _val6(10),
    _val7(11), _val8(12), _val9(13) { } // long member-initializer list

private:
  Int _val1, _val2, _val3, _val4, _val5, _val6, _val7, _val8, _val9;
};

class Test()
{
public:
  Test() { }

private:
  Int _val = 5; // brace-or-equal initializer
};
```

9.4 Class Default Initialization

Class constructors should set initial values, unless otherwise required, due to performance reasons. Pointers must be initialized with `nullptr`. If there is a performance issue, the default initialization should be done as shown in the second example – do it exactly in the same order that the member variables were defined.

```
class Test
{
public:
  Test()
  {
    _index  = NOTOK;
    _memory = nullptr;
    _memorySize = 0;
  }

private:
  Int   _index;
  Char* _memory;
  Int   _memorySize;
};
```

```
class Test
{
public:
  Test(): _index(NOTOK), _memory(nullptr), _memorySize(0)
  {
  }

private:
  Int   _index;
  Char* _memory;
  Int   _memorySize;
};
```

```
class Test
{
public:
  Test():
    _index(NOTOK),
    _memory(nullptr),
    _memorySize(0)
  {
  }

private:
  Int   _index;
  Char* _memory;
  Int   _memorySize;
};
```

9.5 Explicit Constructors

If class constructors support multiple datatypes, use the keyword `explicit` to avoid erroneous auto-conversion. Any constructor with a single argument should use `explicit`.

```
class Test
{
  explicit Test(const String& str);
  explicit Test(Int count);
  explicit Test(Float value);
};
```

9.6 Copy and Assign Operators

Use `MAXON_DISALLOW_COPY_AND_ASSIGN` in any class or struct by default, unless you need to copy your structure. The macro should go at the beginning of the declaration. Add a member CopyFrom instead, which provides safety (as it can return `false`). Classes that contain dynamic data by default should use `MAXON_DISALLOW_COPY_AND_ASSIGN`.

```
class Test
{
  MAXON_DISALLOW_COPY_AND_ASSIGN(Test)

private:
  Int _value[5];
};

class Test
{
  MAXON_DISALLOW_COPY_AND_ASSIGN(Test)

public:
  Bool CopyFrom(const Test& source);
private:
  Int _value[5];
};
```

9.7 Const Parameters

Use the keyword `const` whenever possible, especially for function input parameters and retrieval functions that don't modify a class.

```
void MyFunction(const Vector& a, const String& b);

class MyTest
{
  ...
public:
  Int GetIndex() const;
};
```

9.8 Overrides

Only overload (**Note:** not override!) operators when there is an obvious benefit.

9.9 Structs

Generally classes should be preferred over structs. Use struct only for passive objects that carry data (and if necessary have an Init() function) and offer exclusively public members. If all members of a struct or class are public in the API, they do not need to, but can, carry the underscore. An example is the class Vector.

9.10 Input Parameters by Reference

All input parameters passed by reference must be labeled const.

9.11 Member Layout

Class Members can, but don't have to be, aligned - usually alignment makes sense if the horizontal gaps are relatively small. If there are multiple pointers in one line, the members should be separated.

```
class Test
{
public:
  Int _a;
  Int _b;
  Int _c;
  Char* _d;
};

class Test
{
public:
  Int   _a;
  Int   _b;
  Int   _c;
  Char* _d;
};

class Test
{
public:
  Int*  _a;
  Int*  _b;
  Int*  _c;
  Char* _d;
};

class Test
{
public:
  Int   _a, _b, _c;
  Char* _d;
};
```