EUROPEAN
SPALLATION
SOURCE

On coding conventions, standards and practices

PRESENTED BY MORTEN HILKER-SKAANING

2020-09-14

Naming and casing

Naming and casing

Problems with current coding convention

```
Stats Stats; // compile error

Stats S; // abbreviations not allowed

auto Stats = Stats(); // personally don’t like ‘auto’, undesirable as member definition?

Stats stats; // would prefer this
```

Indifferent to ‘Function’ or ‘function’, but variables should not clash with typenames.

Naming and casing

Standards I’ve seen everywhere

- Constant: `kMyVar`
  - Guarantee no data dependencies on non-constants. Compile-time-only code. Ease refactoring.

Naming and casing

Standards I've seen everywhere

- Constant: `kMyVar`
  - Guarantee no data dependencies on non-constants. Compile-time-only code. Ease refactoring.
- Member variable: `m_MyVar`
  - Distinguish from arguments and local variables. Read data flow into object state. Caution when threading.

Naming and casing

Standards I've seen everywhere

- Constant: `kMyVar`
  - Guarantee no data dependencies on non-constants. Compile-time-only code. Ease refactoring.
- Member variable: `m_MyVar`
  - Distinguish from arguments and local variables. Read data flow into object state. Caution when threading.
- Global/static variable: `g_MyVar`
  - Helps to reason about lifetime and data init order. Caution when threading

Naming and casing

Standards I've seen everywhere

- Constant: `kMyVar`
  - Guarantee no data dependencies on non-constants. Compile-time-only code. Ease refactoring.
- Member variable: `m_MyVar`
  - Distinguish from arguments and local variables. Read data flow into object state. Caution when threading.
- Global/static variable: `g_MyVar`
  - Helps to reason about lifetime and data init order. Caution when threading
- Alternative casing `k_myVar/c_myVar, m_myVar, g_myVar`
- Possibly enforceable via clang-tidy, if we want to do the work.
- Some codebases used SMyStruct (POD), ZMyClass, IMyInterface, not super important

Globals

Globals

Undefined initialization order

Mitigate C++'s global variable undefined initialization order problem.

If you for some reason need complex globals/singletons then wrap in getter function…

Instead of

```
static MyClass g_MyClass;
```

Consider

```
MyClass& GetMyClass() {
  static MyClass g_MyClass;
  return g_MyClass;
}
```

Runtime lib remembers order of function static initialization

Worse when dealing with DynLib/DLL loading and unloading.

Spacing/indent

Spacing/indent

Personal preference for speed reading

```
if (foo (bar))
{
      if (hoge())
      {
            /*Lorem ipsum dolor sit amet, consectetuer adipiscing elit*/
      }
      else if (fuga())
      {
            /*Lorem ipsum dolor sit amet, consectetuer adipiscing elit*/
      }
      else
      {
            /*Lorem ipsum dolor sit amet, consectetuer adipiscing elit*/
      }
}
```

```
MyObject (int a, int b, int c)
: m_a (a)
, m_b (b)
, m_c (c)
{}
```

Spacing/indent

Personal preference for speed reading

space

```
if (foo (bar))
{                 no space
    if (hoge())
    {
        /*Lorem ipsum dolor sit amet, consectetuer adipiscing elit*/
    }
    else if (fuga())
    {
        /*Lorem ipsum dolor sit amet, consectetuer adipiscing elit*/
    }
    else
    {
        /*Lorem ipsum dolor sit amet, consectetuer adipiscing elit*/
    }
}
```

4 “spaces”

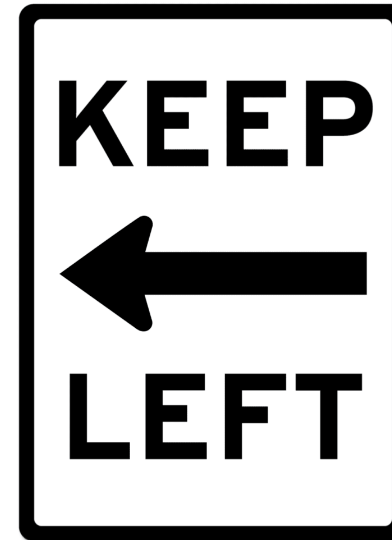
```
MyObject (int a, int b, int c)
: m_a (a)
, m_b (b)
, m_c (c)
{}
```

Spacing/indent

Personal preference for speed reading

1000s of lines? Read only this

```
if (foo (bar))
{
        if (hoge())
        {
                /*Lorem ipsum dolor sit amet, consectetuer adipiscing elit*/
        }
        else if (fuga())
        {
                /*Lorem ipsum dolor sit amet, consectetuer adipiscing elit*/
        }
        else
        {
                /*Lorem ipsum dolor sit amet, consectetuer adipiscing elit*/
        }
}



MyObject (int a, int b, int c)
: m_a (a)
, m_b (b)
, m_c (c)
{}
```



Spacing/indent

Personal preference for speed reading

```
if (foo (bar))
{
        if (hoge(
        {
                /*L
        }
        else if (
        {
                /*L
        }
        else
        {
                /*L
        }
}



MyObject (int
: m_a (a)
, m_b (b)
, m_c (c)
{}
```

1000s of lines? Read only this

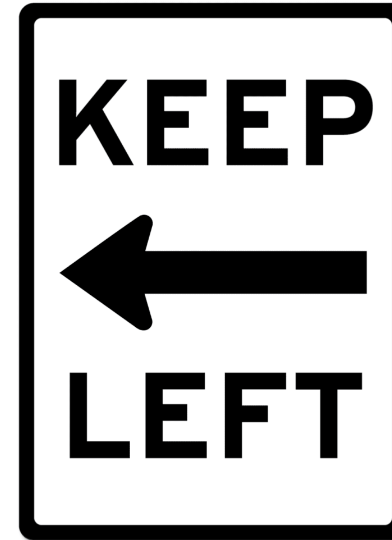
Spacing/indent

Personal preference for speed reading

```
if (foo (bar))
{
        if (hoge(
        {
                /*L
        }
        else if (
        {
                /*L
        }
        else
        {
                /*L
        }
}



MyObject (int
: m_a (a)
, m_b (b)
, m_c (c)
{}
```

```
if (foo(bar))
        if (hoge(
                /*L
        } else if
                /*L
        } else {
                /*L
        }
}



MyObject(int a
```

clang-format:
Structure is lost
Have to read RHS



Unsorted practices

Unsorted practices

- Split into modules: Logically different systems (sound, physics, render, ui) walled off with interfaces. Exchange data as PODs or "pure" interface objects
  - Never exposes 3rdparty code/middleware to other systems.
  - Optimizes compilation and linking, and to be able to "discard" systems.
  - Could also try Pimpl-like classes for this

Unsorted practices

- Split into modules: Logically different systems (sound, physics, render, ui) walled off with interfaces. Exchange data as PODs or “pure” interface objects
  - Never exposes 3rdparty code/middleware to other systems.
  - Optimizes compilation and linking, and to be able to “discard” systems.
  - Could also try Pimpl-like classes for this

- Create local structs to hold context for computation with **many** objects
  - Sign class is too big?

Unsorted practices

- Split into modules: Logically different systems (sound, physics, render, ui) walled off with interfaces. Exchange data as PODs or "pure" interface objects
  - Never exposes 3rdparty code/middleware to other systems.
  - Optimizes compilation and linking, and to be able to "discard" systems.
  - Could also try Pimpl-like classes for this
- Create local structs to hold context for computation with **many** objects
  - Sign class is too big?
- A byte-stream wrapper that provides read/writeFloat(), read/writeUint64(), read/writeString() etc for types with alignment
  - Useful for networking/serialization

Unsorted practices

- Split into modules: Logically different systems (sound, physics, render, ui) walled off with interfaces. Exchange data as PODs or “pure” interface objects
  - Never exposes 3rdparty code/middleware to other systems.
  - Optimizes compilation and linking, and to be able to “discard” systems.
  - Could also try Pimpl-like classes for this
- Create local structs to hold context for computation with **many** objects
  - Sign class is too big?
- A byte-stream wrapper that provides read/writeFloat(), read/writeUint64(), read/writeString() etc for types with alignment
  - Useful for networking/serialization
- ONLY_ON_BRANCH(BranchName)
  - Macros to allow code to be compiled on a named branch. Or several of branches. Same for disallow.

Unsorted practices

- Split into modules: Logically different systems (sound, physics, render, ui) walled off with interfaces. Exchange data as PODs or "pure" interface objects
  - Never exposes 3rdparty code/middleware to other systems.
  - Optimizes compilation and linking, and to be able to "discard" systems.
  - Could also try Pimpl-like classes for this
- Create local structs to hold context for computation with **many** objects
  - Sign class is too big?
- A byte-stream wrapper that provides read/writeFloat(), read/writeUint64(), read/writeString() etc for types with alignment
  - Useful for networking/serialization
- ONLY_ON_BRANCH(BranchName)
  - Macros to allow code to be compiled on a named branch. Or several of branches. Same for disallow.
- Testing: "Negated tests"
  - A parameter telling your test that it must fail, so you can test that tests are fallible, which is required to be useful.

Finish presentation