Methodology for Freestanding Development.

Amlal El Mahrouss.
amlal@nekernel.org

December 2025

Abstract

Many low-level software have been shipped using the C programming language. And some of them, such as EKA2 use the C++ programming language. Although notoriously difficult, one may adapt to those constraints in order to deliver one such operating system kernel. This is why most production-grade software (Linux, XNU, and NT) are mostly written in C. With a higher-level subset in C++. However, when correctly applying the following principles to freestanding development, it becomes much easier to ensure correctness of such programs.

1 The Three Principles of Freestanding Development.

1.1 I: The Run-Time Domain.

The problem lies in the programming language runtime, which assumes an existing host. The contrary of a hosted environment is freestanding, a computing mode which doesn't expect a hosted runtime. Such programs may use the compile-time evaluation domain to achieve minimal run-time domain usage.

1.2 II: The Compile-Time Evaluation Domain.

One may avoid Virtual Method Tables or a runtime when possible. While focusing instead on meta-programming and compile-time features offered by C++. For example one may use templates to implement a scheduling policy algorithm. One example of such implementation may be:

```
struct FileTree final {
    static constexpr bool is_virtual_memory = false;
    static constexpr bool is_memory = false;
    static constexpr bool is_file = true;
    /// ...
};

struct MemoryTree final {
    static constexpr bool is_virtual_memory = false;
    static constexpr bool is_memory = true;
    static constexpr bool is_file = false;
    /// ...
};
```

Source: Link.
Which is why the 'constexpr' keyword is very powerful here for the Compile-Time Evaluation Domain, we avoid the many pitfalls of the Run-Time Evaluation Domain.

1.3 III: Memory Layout and the example of C++.

The Virtual Method Table (now defined as the VMT) is a big part of the problem, one may illustrate the following:

```cpp
/// /std:c++20 /Wall

#include <iostream>

class A
{
public:
    explicit A() = default;
    virtual ~A() = default;

    virtual void doImpl()
    {
        std::cout << "doImpl()\r\n";
    }
};

class B : public A
{
public:
    explicit B() = default;
    ~B() override = default;
};

int main() {
    B callImpl;
    callImpl.doImpl();
}
```

Source: Link.
The following can instead be done to achieve similar results using the Compile-Time Evaluation Domain.

```cpp
inline constexpr auto kInvalidType = 0;

template <class Driver>
concept IsValidDriver = requires(Driver drv) {
  { static_assert(drv.IsActive() && drv.Type() > kInvalidType, "Driver is not valid for usage.") };
};
```

Source: Link.
Now, the problem with freestanding development is that such feature may be abused, and it is mitigated by following the TTPI.

1.4 IV: The Three Prongs on Inheritance Decision Framework.

The TTPI is a boolean framework used to evaluate whether one may consider using a Object Oriented programming language inside a freestanding program, consider the following:

1: Is this implementable with compile-time protocols/concepts?

2: Is this implementable without three trade-off costs?
    Without violating the Runtime cost?
    The Verification cost?
    The Known-Ahead-Correctness cost?

3: Is this implementable without using a VMT?

If two of the three conditions fail, then the framework fails, and you should consider finding another solution to your problem as it surely has an equivalent without the problematic aspects.

1.5 V: Compile-Time Vetting in a Freestanding Domain.

The following concept makes sure that the 'class T' is vetted by the domain. Such properties are called 'Vettable' such program in the domain makes sure that a 'Container' is truly deemed fit for a Run-Time or Compile-Time Evaluation Domain. The 'Vettable' structure makes use of template meta-programming in C++ to evaluate whether a 'Container' shall be vetted. Such system may look as such in a Compile-Time Evaluation Domain:

```
#define NE_VETTABLE static constexpr BOOL kVettable = YES;
#define NE_NON_VETTABLE static constexpr BOOL kVettable = NO;

template <class Type>
concept IsVettable = requires(Type) {
  (Type::kVettable);
};

/// This structure is vettable.
struct UnVettable {
  NE_VETTABLE;
};

/// This structure is unvettable.
struct UnVettable {
  NE_NON_VETTABLE;
};

/// One example of a usage.
if constexpr (IsVettable<UnVettable>) {
  instVet->Vet();
} else {
  instVet->Abort();
}
```

Source: Link.

2 VI: Conclusion

Safe and correct development in a freestanding domain is indeed possible granted the above concepts are applied and respected.

3 References

1. NeKernel.org (2025). NeKernel Operating System. Available at: https://nekernel.org
2. Sales, J., Tasker, M. (2005). Introducing EKA2. Symbian OS Internals. Wiley. Available at: https://media.wiley.com/product_data/excerpt/47/04700252/0470025247.pdf
3. Driesen, K., Hölzle, U. (1996). The direct cost of virtual function calls in C++. ***OOPSLA '96***. ACM. DOI: 10.1145/236338.236369