

Methodology for Freestanding Development.

Amlal El Mahrouss
amlal@nekernel.org

December 2025
Last Edited: January 2026

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 Evaluation 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:

```
1 struct FileTree final {
2     static constexpr bool is_virtual_memory = false;
3     static constexpr bool is_memory = false;
4     static constexpr bool is_file = true;
5     /// ...
6 };
7
8 struct MemoryTree final {
9     static constexpr bool is_virtual_memory = false;
10    static constexpr bool is_memory = true;
11    static constexpr bool is_file = false;
12    /// ...
13 };
```

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:

```
1  /// /std:c++20 /Wall
2
3  #include <iostream>
4
5  class A
6  {
7  public:
8      explicit A() = default;
9      virtual ~A() = default;
10
11     virtual void doImpl()
12     {
13         std::cout << "doImpl()\r\n";
14     }
15 };
16
17 class B : public A
18 {
19 public:
20     explicit B() = default;
21     ~B() override = default;
22 };
23
24 int main() {
25     B callImpl;
26     callImpl.doImpl();
27 }
```

Source: [Link](#).

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

```
1  inline constexpr auto kInvalidType = 0;
2
3  template <class Driver>
4  concept IsValidDriver = requires(Driver drv) {
5      { drv.IsActive() && drv.Type() > kInvalidType };
6  };
```

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.

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?

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

The following concept makes sure that the ‘class T’ is vetted by the domain. Such properties are called ‘Vetable’ such program in the domain makes sure that a ‘Container’ is truly deemed fit for a Run-Time or Compile-Time Evaluation Domain. The ‘Vetable’ 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:

```
1 #define NE_VETTABLE static constexpr BOOL kVetable = YES;
2 #define NE_NON_VETTABLE static constexpr BOOL kVetable = NO;
3
4 template <class Type>
5 concept IsVetable = requires(Type) {
6     (Type::kVetable);
7 };
8
9 /// This structure is vettable.
10 struct Vetable {
11     NE_VETTABLE;
12 };
13
14 /// This structure is unvettable.
15 struct UnVetable {
16     NE_NON_VETTABLE;
17 };
18
19 /// One example of a usage.
20 if constexpr (IsVetable<UnVetable>) {
21     instVet->Vet();
22 } else {
23     instVet->Abort();
24 }
```

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