# Machine-checked object layout for C++ multiple inheritance with empty-base optimization

Tahina Ramananandro
http://gallium.inria.fr/~tramanan/cpp/object-layout

August 6, 2010

## Notations and conventions

We adopt the usual Coq list notations:

- $\mathsf{nil}$ is the empty list
- $a :: q$ is the list starting with an element $a$ and continuing with the tail list $q$
- $+\!\!+$ is the *append* (list concatenation) operator: for any list $l$, we have $\mathsf{nil} +\!\!+ l \underset{\text{def.}}{=\!=} l$ and $(a :: q) +\!\!+ l \underset{\text{def.}}{=\!=} a :: (q +\!\!+ l)$.

We also adopt the following additional notations:

- $\mathsf{first}$ is a function defined on non-empty lists, such that $\mathsf{first}(a :: l') \underset{\text{def.}}{=\!=} a$ for all $a, l'$.
- $\mathsf{last}$ is a function defined on non-empty lists, computing their last elements: $\mathsf{last}(a :: \mathsf{nil}) \underset{\text{def.}}{=\!=} a$ and $\mathsf{last}(a :: b :: l') \underset{\text{def.}}{=\!=} \mathsf{last}(b :: l')$ for all $a, b, l'$.
- $\mathsf{length}(l)$ is the length of a list $l$: $\mathsf{length}(\mathsf{nil}) \underset{\text{def.}}{=\!=} 0$ and $\mathsf{length}(a :: l') \underset{\text{def.}}{=\!=} 1 + \mathsf{length}(l')$
- For any sets $\mathcal{A}$ and $\mathcal{S} \subseteq \mathcal{A} \times \mathbb{Z}$, and any $\omega \in \mathbb{Z}$:

$$\omega + \mathcal{S} \underset{\text{def.}}{=\!=} \{(A, \omega + o) \mid (A, o) \in \mathcal{S}\}$$

- For any two sets $\mathcal{S}, \mathcal{T}$: $\mathcal{S} \bot \mathcal{T}$ stands for $\mathcal{S} \cap \mathcal{T} = \varnothing$
- for any set $\mathcal{S}$, we pose

$$\mathsf{option}\ \mathcal{S} \underset{\text{def.}}{=\!=} \{\varnothing\} \cup \{\{S\} \mid S \in \mathcal{S}\}$$

  which is the set of all subsets of $\mathcal{S}$ with cardinality 0 or 1.

- for any two integers $u, v$: $(u \mid v)$ if and only if $u$ *evenly divides* $v$, i.e. $v$ is a multiple of $u$:

$$(u \mid v) \Leftrightarrow \exists k : \ v = k \cdot u$$

Our examples are given either by GNU GCC 4.3 on an Intel 32-bit machine architecture (where **sizeof(int)** = 4) with option `-malign-double` enabled, or by the prescriptions of Itanium C++ ABI [1].

# Contents

# 1 Goal

Our work aims at:

- formalizing the object-oriented part of C++ with multiple inheritance and structure arrays
- finding realistic implementation criteria for C++ object layout
- proving that any implementation matching our criteria is correct with respect to our formalization

Our work is machine-checked with the Coq proof assistant.
More context information about the motivation of our work can be found in our POPL 2011 draft [6].

# 2 The semantics of C++ multiple inheritance

We started from [9] who formalized C++ multiple inheritance in Isabelle based on [7]. We extend their work by featuring structures and structure arrays.

## 2.1 Class hierarchy (`Cplusconcepts.v`)

**Notation 1.** *Let $\mathcal{A}$ be the set of* atomic types *($\mathsf{int}, \mathsf{float}, \mathsf{short}, \ldots$).*

We make no further assumption on atomic types or values: our formalization is a Coq functorial module taking them as parameters.

**Definition 1.** *A* class hierarchy *is a tuple $(\mathcal{C}, \mathcal{FI}, \mathcal{SCF}, \mathcal{STF}, \mathcal{DNV}, \mathcal{DV})$, where:*

- *$\mathcal{C}$ is the set of* classes. *We assume $\mathcal{C} \cap \mathcal{A} = \varnothing$.*
- *$\mathcal{FI}$ is the set of* field identifiers
- *$\mathcal{SCF} : \mathcal{C} \to \mathfrak{P}(\mathcal{FI} \times (\mathcal{A} \cup \mathcal{C}))$ gives, for each class, the set of its* scalar fields *(a scalar field is either an atom, or a pointer to $C$ for some class $C$).*
- *$\mathcal{STF} : \mathcal{C} \to \mathfrak{P}(\mathsf{StructField})$ (where $\mathsf{StructField} \underset{\text{def.}}{=\!=} \mathcal{FI} \times \mathcal{C} \times \mathbb{N}^*$)[1] gives, for each class, the set of its* structure array fields. *Each structure array field $(f, C, n)$ is considered to be an array of $n$ structures of type $C$. In fact, structure fields are structure array fields where $n = 1$.*

[1] The Standard forbids arrays with zero cells.

- $\mathcal{DNV} : \mathcal{C} \to \mathfrak{P}(\mathcal{C})$ *gives, for each class, the set*[2] *of its non-virtual direct bases.*
- $\mathcal{DV} : \mathcal{C} \to \mathfrak{P}(\mathcal{C})$ *gives, for each class, the set of its virtual direct bases.*

For instance, the following code:

```
struct A             { int i; };
struct B: virtual A  { C * c; };
struct C: A, B       { float j; A a[2]; B b; };
```

would give the following hierarchy:

| | | |
|---|---|---|
| $\mathcal{C}$ | = | $\{A, B, C\}$ |
| $\mathcal{FI}$ | = | $\{i, j, a, b, c\}$ |
| $\mathcal{ScF}$ | : | $A \mapsto \{(i, \mathsf{int})\}$<br>$B \mapsto \{(c, C)\}$<br>$C \mapsto \{(j, \mathsf{float})\}$ |
| $\mathcal{StF}$ | : | $A \mapsto \varnothing$<br>$B \mapsto \varnothing$<br>$C \mapsto \{(a, A, 2), (b, B, 1)\}$ |
| $\mathcal{DNV}$ | : | $A \mapsto \varnothing$<br>$B \mapsto \varnothing$<br>$C \mapsto \{A, B\}$ |
| $\mathcal{DV}$ | : | $A \mapsto \varnothing$<br>$B \mapsto \{A\}$<br>$C \mapsto \varnothing$ |

If a Coq formalization had to exactly mimic these definitions, then it would need dependent types, as the domain of, say, DNV depends on $\mathcal{C}$, both being part of the hierarchy data. But we chose to avoid dependent types to represent those functions and families, by using finite mappings and externally imposing *well-formedness* constraints (`CplusWf.v`).
In practice, Compcert provides a library for finite mappings (`Maps.v`), to represent functions $f : \mathbb{N}^* \to \mathsf{option}\ \mathcal{S}$ such that the "domain" $\{n \in \mathbb{N}^* \mid f(n) \neq \varnothing\}$ is finite. For any set $\mathcal{S}$, the set of such finite mappings is denoted PTree.t $\mathcal{S}$ in respect to the tree implementation.
In our case, we do not explicitly define the set $\mathcal{C}$ of the classes of a hierarchy. Instead, we define (`Cplusconcepts.v`) a hierarchy to be a mapping $h \in$ PTree.t Class.t where Class.t is a record type having fields such as $\mathcal{DNV}, \mathcal{DV}, \ldots$ Then, class identifiers are in $\mathbb{N}^*$ and, morally, $\mathcal{C}$ is the "domain" of $h$, that is the set of class identifiers $C$ such that $h(C) \neq \varnothing$.
But we also avoid dependent types for the value types of the fields of Class.t. For instance, we have, for any class $C$ such that $h(C) = \{c\}$ for some $c \in$ Class.t, $c.\mathcal{DNV} \in$ list $\mathbb{N}^*$ (and not list $\mathcal{C}$, which would require dependent

[2] For our purpose, direct base ordering is not relevant.

types, as $\mathcal{C}$ depends on $h$). To enforce the domains, we externally require *completeness* conditions (`CplusWf.v`) such as:

$$\forall C, c : h(C) = \{c\} \Rightarrow \forall b \in c.\mathcal{DNV}, h(b) \neq \varnothing$$

which states that every direct non-virtual base of a class is a class belonging to the hierarchy. (More precisely, every identifier of the set of direct non-virtual base identifiers is a valid identifier of an existing class.)

## 2.2 Inheritance paths, or Base class subobjects (Cpluscon-cepts.v)

### 2.2.1 Non-virtual inheritance

Consider the following code:

```
struct A             { int a; };
struct B1: A         {};
struct B2: A         {};
struct C : A         {};
struct D : B1, B2, C {};
```

Then, an instance of $D$ will have *three* different "copies" of $A$: one reachable through the direct non-virtual base $B1$, one through $B2$, and one from $C$. This is called *non-virtual inheritance*, or *repeated inheritance*. Each "copy", called a *subobject* of $D$ of static type $A$, has its own value for the field $a$. So, it is necessary to distinguish those subobjects. Following Wasserrab et al., those subobjects can be distinguished by the *paths* through which they are reached.
More precisely:

**Definition 2.** *A list $l$ is a* non-virtual path *from $C$ to $A$ if, and only if:*

- *either $C = A$ and $l = A :: nil$. This path is called the* trivial path.
- *or there exists a non-virtual direct base $B$ of $C$ and a non-virtual path $l'$ from $B$ to $A$, such that $l = C :: l'$.*

*$A$ is a* non-virtual base *of $C$ if, and only if, $A$ is reachable through a non-trivial* non-virtual path *from $C$.*

Then, in the above example, $A$ is a non-virtual base of $C$ through two *different* paths: $C :: B1 :: A :: \mathsf{nil}$ and $C :: B2 :: A :: \mathsf{nil}$. But B2 is not a non-virtual base of B1.
Then, if we have

```
C    c;
B1 * b1 = (B1 *) &c;
B2 * b2 = (B2 *) &c;
A  * a1 = (A  *) b1;
A  * a2 = (A  *) b2;
```

the C++ Standard [4] dictates $a_1 \neq a_2$: even though they are pointers to subobjects of the same $C$ object, those subobjects are accessible through *different non-virtual paths*, so those pointers must be different.
This is one of the properties our work aims at proving.
In particular:

**Lemma 1.** *Any non-virtual path from any class $C$ to any class $B$ begins with $C$ and ends with $B$.*

which leads to the:

**Lemma 2.** *If $C :: l$ is a non-virtual path from $C$ to $B$ and $B :: l'$ is a non-virtual path from $B$ to $A$, then $C :: l{+}{+}l'$ is a non-virtual path from $C$ to $A$ denoted $(C :: l)@_{\mathsf{Repeated}}(B :: l')$.*

### 2.2.2 Virtual inheritance

C++ features *virtual inheritance*, to allow some subobjects accessible through apparently different paths to be equal.Rossie and Friedman [7] (formalized in Isabelle by Wasserrab, Nipkow et al. [9]) give a notion of path allowing such subobjects to be represented with actually *the same* path.

**Definition 3.** *A class $A$ is a* virtual base *of $C$ if, and only if:*

- *either $A$ is a direct virtual base of $C$*
- *or there is a direct (virtual or non-virtual) base $B$ of $C$ such that $A$ is a virtual base of $B$.*

**Definition 4.** *A* path, inheritance path, *or* base class subobject, *from a class $C$ to a class $A$ is a couple $(h, l)$ such that:*

- *either $h = \mathsf{Repeated}$ and $l$ is a non-virtual path from $C$ to $A$*
- *or $h = \mathsf{Shared}$ and there exists a virtual base $B$ of $C$ such that $l$ is a non-virtual path from $B$ to $A$.*

*The set of all paths shall be denoted $\mathsf{Path}$.*

For instance, if we have:

```
struct A                {};
struct B1: virtual A    {};
struct B2: virtual A    {};
struct C : A            {};
struct D : A, B1, B2, C {};
```

Then, the path from D to A through B1 denotes the same subobject as the path through B2, because A is a *virtual* base of both B1 and B2: the A subobject is *shared* by both classes. As B1 and B2 are bases of D, then A is a virtual base of D and this subobject is denoted by the path $(\mathsf{Shared}, A :: \mathsf{nil})$. However, A is also accessible from D through two non-virtual paths: $D :: A :: \mathsf{nil}$ (because A is a non-virtual direct base of D), and $D :: C :: A :: \mathsf{nil}$. Those paths denote three different subobjects.

**Lemma 3.** *Let $C_0, C_1, C_2$ be three classes. If $(h_i, l_i)$ is a path from $C_{i-1}$ to $C_i$ for each $i \in 1, 2$, then:*

- *if $h_2 = \mathsf{Repeated}$, then $(h_1, l_1 @_{\mathsf{Repeated}} l_2)$ is a path from $C_0$ to $C_2$*
- *if $h_2 = \mathsf{Shared}$, then $(h_2, l_2)$ is a path from $C_0$ to $C_2$*

*This path is denoted $(h_1, l_1)@(h_2, l_2)$.*

### 2.2.3 A practical example

Non-virtual and virtual multiple inheritance are not uncommon in everyday life. Consider for instance three electronic devices: a **radio**, an **alarm**, and a **radio-alarm**.

- A *radio* needs electronic current provided by a **plug**. It has a **switch** to turn it on or off. The radio station currently running can be controlled via its **frequency**.
- An *alarm* needs electronic current provided by a **plug**. It has a **switch** to turn it on or off. The **wake-up time** tells when the alarm should ring up.
- A *radio-alarm* combines both a **radio** and **alarm**. Either the **radio** and the **alarm** can be turned on or off independently of the other, through the corresponding **switch**. By contrast, the radio-alarm also needs electric current provided by a **plug** common to both radio and alarm components.

A radio-alarm device could be modelled by a C++ program using multiple inheritance, and especially both non-virtual and virtual inheritance.

```
struct Plug {
  bool plugged;
}

struct Device: virtual Plug {
  bool switch;
}

struct Radio: Device {
```

```
  double frequency;
}

typedef double GNUEpoch;

struct Alarm: Device {
  GNUEpoch wakeUpTime;
  virtual void ring ();
}

struct RadioAlarm: Radio, Alarm {
  void ring () {
    if (Radio::switch) {
      Alarm::ring ();
    } else {
      Radio::switch = true;
    }
  }
}
```

Here, a RadioAlarm will have one frequency for its Radio, one wakeUpTime for its Alarm, two **switch**es (one for its Radio, one for its Alarm), but only one plugged status. It is impossible to unplug the Radio without unplugging the Alarm: they share the same plug.
The Alarm has its own method to ring up, usually by buzzing. This method is called at timeout, through the following code:

```
GNUEpoch getTime ();

void wait (Alarm * alarm) {
  if    (getTime () > alarm->wakeUpTime) return;
  while (getTime () < alarm->wakeUpTime) {};
  alarm->ring ();
}
```

But a RadioAlarm may want to turn the radio on instead of buzzing, unless the radio is already turned on. That is why the RadioAlarm class *redefines*, or *overrides*, the method ring, so that the following code:

```
int main (int, String[]) {
  RadioAlarm a;
  wait (static_cast<Alarm*>(&a));
  return 0;
}
```

will actually call the ring method defined in class RadioAlarm. To enable this, ring must be declared **virtual** in Alarm.

### 2.3 Structure array fields

Our work extends Wasserrab, Nipkow et al. with *structure array fields*. That is, *whole objects* can be embedded in a given structure field or structure array field. Keeping this point of view would break the condition of two whole objects to be disjoint, so it is necessary to take structure fields into account by formalizing a notion of *generalized subobjects* including structure field access, not only inheritance.
Our work features arrays, but only considers structure arrays. Indeed, arrays of scalars can be simulated through arrays of structures having only one scalar field. For instance[3]:

```
int * t = new int[18];
t[4] = 42;
```

can be simulated with:

```
struct Sint { int value; };
Sint * t = new Sint()[18];
t[4].value = 42;
```

This point of view has no significant impact on performance[4]: accessing the field of a structure from a structure array yields only one memory access, as many as accessing a field from an array of scalars.

Any non-null object pointer $b$ of type $B*$ evaluates to a pointer of the following (informal) form:

$$\begin{array}{llcl} C_1 & * & c_1 & = & ((B_1*)\&c_0[i_1]) \rightarrow f_1 \\ C_2 & * & c_2 & = & ((B_2*)\&c_1[i_2]) \rightarrow f_2 \\ & & \vdots & & \\ C_k & * & c_k & = & ((B_k*)\&c_{k-1}[i_k]) \rightarrow f_k \\ B & * & b & = & (B*)\&c_k[i] \end{array}$$

where:

- $c_0$ is a structure array of some class $C_0$ of size $n_0$
- $0 \leq k$
- for each $j \in [1 \ldots k]$, $0 \leq i_j < n_{j-1}$
- for each $j \in [1 \ldots k]$, there is an inheritance path $P_j$ from $C_{j-1}$ to some class $B_j$

[3] This simulation pattern works not only for atomic types, but also for pointer types. A generic pattern for scalar types would use a template. However, templates are out of the scope of our work.

[4] As long as Sint has only a default constructor, which may be ignored at execution. This is out of the scope of our work.

- for each $j \in [1 \dots k]$, there is a class $C_j$ such that $f_j$ is an structure array field of $B_j$ of type $C_j[n_j]$ for some $n_j$
- $0 \leq i < n_k$
- there is an inheritance path $P$ from $C_k$ to $B$

We call the sequence $(i_1, P_1, f_1); (i_2, P_2, f_2); \dots; (i_k, P_k, f_k)$ an *array path from* $C_0[n_0]$ *to* $C_k[n_k]$. Let us denote it $\mathcal{P}$.
More formally:

**Definition 5.** *Let* $C, C' \in \mathcal{C}$, $n, n' \in \mathbb{N}$ *and* $l$ *be a list of elements of the set* $\mathbb{N} \times \mathit{Path} \times \mathit{StructField}$. *We say that* $l$ is an array path from $C[n]$ to $C'[n']$ *if, and only if, one of these conditions holds:*

- $C = C'$ *and* $n' \leq n$ *and* $l = \mathit{nil}$
- *or all the following conditions hold:*
  - *there exists* $i \in \mathbb{N}$ *such that* $i < n$
  - *there is a class* $A \in \mathcal{C}$ *and a path* $p \in \mathit{Path}$ *from* $C$ *to* $A$
  - *there is a structure array field* $F = (f, D, s) \in St\mathcal{F}(A)$
  - *there is a list* $l'$ *such that* $l'$ *is an array path from* $D[s]$ *to* $C'[n]$
  - *and* $l = (i, p, F) :: l'$

**Definition 6.** *A* generalized subobject *or* relative pointer $p$ *of static type* $A$ *relatively to an array of type* $C[n]$ *(or a* relative pointer from $C[n]$ to $A$*)* *is a triple* $(\mathcal{P}, i, P)$ *where:*

- $\mathcal{P}$ *is an array path from* $C[n]$ *to some* $C'[n']$
- $i$ *is an index in the array of type* $C'[n']$, *such that* $0 \leq i < n'$
- *and* $P$ *is an inheritance path from* $C'$ *to* $A$

**Definition 7.** *A run-time pointer* $\pi$ *of static type* $A$ *is a couple* $(c, p)$ *where:*

- $c$ *is the initial structure array, either statically declared, or dynamically created, of some type* $C[n]$
- $p$ *is a relative pointer from* $C[n]$ *to* $A$

For instance, if we have the following code:

```
struct A0            {};
struct A : A0        {};
struct B             { A a[4]; };
struct C1: virtual B {};
struct C2: virtual B {};
struct D : C1, C2    {};
struct E             { D d[5]; };
E e[7];
```

then the expression:

```
(A0 *) &(e[2].d[0].a[3])
```

yields the pointer:

$$
\begin{array}{l}
(\\
\quad e,\\
\quad (2, (\mathsf{Repeated}, E :: \mathsf{nil}), d) :: (0, (\mathsf{Shared}, B :: \mathsf{nil}), a) :: \mathsf{nil},\\
\quad 3,\\
\quad (\mathsf{Repeated}, A :: A0 :: \mathsf{nil})\\
)
\end{array}
$$

where $(2, (\mathsf{Repeated}, E :: \mathsf{nil}), d) :: (0, (\mathsf{Shared}, B :: \mathsf{nil}), a) :: \mathsf{nil}$ is an array path from $E[7]$ to $A[4]$.
This apparently counterintuitive presentation allows us however to easily formalize object pointer arithmetics. Indeed, any *full object* (that is, any subobject that is not the result of a non-trivial inheritance cast) actually always belongs to some structure array (either initial or defined by a structure array field). So, if $p$ is a pointer to such a full object, then $p[q]$ may be defined (not necessarily for any $q$ however).
As $p$ points to a full object, we have $B = C_k$, so the path $P$ is trivial, and we have:

$$p = \&(c[i_1].f_1[i_2].f_2 \ldots .f_{k-1}[i_k].f_k[i]) = (c, \mathcal{P}, i, P)$$

so that, for any $q$ such that $0 \leq i_{k+1} + q < n_k$:

$$p[q] = \&(c[i_1].f_1[i_2].f_2 \ldots .f_{k-1}[i_k].f_k[i+q]) = (c, \mathcal{P}, i+q, P)$$

That is, the only change is the index in the array of type $C_k[n_k]$. In particular, the array path and the inheritance path remain unchanged.
Our idea also allows to easily express the other two basic operations on pointers:

- if $p = (c, \mathcal{P}, i, P)$ is a pointer to some subobject of type $B$ and if $Q$ is an inheritance path from $B$ to $A$ such that derived-to-base casting of a pointer to $B$ yields a pointer to $A$ using $Q$, then $(A*)p = (c, \mathcal{P}, i, P@Q)$. Similarly for base-to-derived casting, if $p = (c, \mathcal{P}, i, P@Q)$, then base-to-derived casting would lead to $(c, \mathcal{P}, i, P)$.

- if $p = (c, \mathcal{P}, i, P)$ is a pointer to some subobject of type $B$ and if $f$ is a structure field of type $A$ defined in class $B$, then $\mathsf{p{-}{>}f} = (c, (\mathcal{P} +\!\!+ (i, P, f) :: \mathsf{nil}, 0, Q))$ where $Q = (\mathsf{Repeated}, A :: \mathsf{nil})$ is the trivial inheritance path from $A$ to $A$. In other words, $\mathsf{p{-}{>}f}$ points to the first structure of the array defined by $f$.

## 2.4 Well-founded hierarchies (`CplusWf.v`)

We expect some well-foundedness hypotheses to hold on the class hierarchy, so as to be able to *compute*, for any class $C$, some data $F(C)$ depending on $F(B)$ for all classes $B$ being bases of $C$ or types of the structure array fields of $C$.

**Notation 2.** *For any class $C$, $\mathcal{D}(C)$ shall denote the set of its direct bases (virtual or non-virtual):*

$$\mathcal{D}(C) \underset{\text{def.}}{=\!=} \mathcal{DNV}(C) \cup \mathcal{DV}(C)$$

**Hypothesis 1.** *Assume the hierarchy is* well-founded*, i.e. the set of classes can be ordered by some well-founded relation $\prec$ compatible with inheritance, i.e. such that:*

$$\forall B \in \mathcal{D}(C) : \ B \prec C$$

*and compatible with structure array fields, i.e. such that:*

$$\forall (f, B, n) \in St\mathcal{F}(C) : \ B \prec C$$

In our Coq formalization, $\mathcal{C} \subseteq \mathbb{N}^*$, and $\prec$ is the usual ordering relation on $\mathbb{N}^*$.

**Notation 3.** *For each class $C$, the set of its virtual bases is denoted $\mathcal{V}(C)$. Thus, as the hierarchy is well-founded, we can compute $\mathcal{V}(C)$ by well-founded induction over $\prec$:*

$$\mathcal{V}(C) \underset{\text{def.}}{=\!=} \mathcal{DV}(C) \cup \bigcup_{B \in \mathcal{D}(C)} \mathcal{V}(B)$$

**Lemma 4.**

$$\forall C, \forall B \in \mathcal{V}(C) : \ B \prec C$$

*In particular, $C \notin \mathcal{V}(C)$.*

**Lemma 5.** *If there exists a path from $C$ to $A$, then $A \preceq C$.*

**Lemma 6.** *The only path from a class $C$ to itself is the* trivial *path:*

$$(\mathit{Repeated}, C :: \mathit{nil})$$

**Lemma 7.** *For any classes $C$ and $A$, if $(h, B :: l)$ is a path from $C$ to $A$, then $B = C$ if and only if $h = \mathit{Repeated}$.*

# 3 Formalization of object layout

In this section, the hierarchy is assumed to be well-founded.
Our goal is to compute, for any pointer $p$ to a subobject of type $A$, a memory offset:

- such that if $q$ is a pointer to a different subobject of type $A$, then $p$ and $q$ must have different memory offsets
- and, for any scalar field $f$ defined in the class $A$, the offset in memory where the value of `p->f` is stored, such that any two different *scalar* fields never overlap. This allows showing the *good variable property* on concrete memory models such as Compcert.

Those two goals do not necessarily entail each other. Indeed, if $A$ has no fields, then pointers to distinct subobjects of type $A$ still have to be distinguished, which forbids structures from occupying zero-sized storage. To avoid waste of space, most compilers introduce *empty base optimizations* to allow empty structures of *different* types to overlap.
Our aim is to prove the correctness of a family of layout algorithms. So, we first give soundness conditions, which we prove to be correct with respect to the two

## 3.1 Methodological overview

Our work formalizes a layout principle common to most current C++ compilers.

### 3.1.1 The dynamic type data of a subobject (or "pointer to virtual table")

When a virtual method is called from a subobject of some type $A$, most compilers use specific data such as a virtual table to call the method. This data is specific to the class of the whole object of the considered subobject. Indeed, the non-virtual structure of a subobject of type $A$ is independent on how the subobject is reached: for any field of $A$ (or of a non-virtual base of $A$), the offset to access this field will be always the same regardless of how the subobject is reached. $A$ is the *static type* of the subobject. However, calling a method, or reaching a virtual base, depends on the way the subobject is reached: this additional information is the *dynamic type* of the subobject. At the semantics level, it is actually the inheritance path from the full object to the subobject.
At the implementation level, most C++ compilers introduce additional data at the beginning of some classes to store that dynamic type data. For instance, GNU GCC stores a pointer to the virtual table of the subobject, which is different depending on the dynamic type of the subobject. But

other compilers may use other additional data than virtual tables (for instance a key for dictionary search). Our formalization does not depend on the actual data stored, it only fixes the size.
In practice, such additional data are needed if the class has or inherits a virtual method, or has a virtual base. Such class is called *dynamic*.
However, our formalization of object layout does not rely on a precise definition of this notion, which is required only for concrete algorithms.

### 3.1.2 Layout scheme

We formalize the following layout for a full instance of some class $C$, common to many C++ compilers:

- The non-virtual part of $C$, consisting of:
  - The non-virtual part of a dynamic non-virtual direct base $A$ of $C$, if such class exists
  - If $C$ is dynamic but such $A$ does not exist, then some space for the *dynamic type data* for $C$
  - Then the non-virtual parts of other non-virtual direct bases of $C$
  - Then the fields of $C$, so that they never overlap
- Then the non-virtual parts of all virtual bases of $C$, so that they never overlap

### 3.1.3 The need for an empty base optimization

Empty classes must have a nonzero size. Indeed, if for instance **sizeof**(A) = 0, then two objects from two distinct indices of an array of $A$ could not be distinguished.
So assume **sizeof**(A) = 1. Then, it would be sound to have **sizeof**(B1) = 1 and **sizeof**(B2) = 1 for classes $A, B_1, B_2$ below:

```
struct A          {};
struct B1: A      {};
struct B2: A      {};
struct C : B1, B2 { int i; };
```

A naive compiler would lay out $B_1$, $B_2$ and $i$ in disjoint memory locations, yielding **sizeof**(C) >= 2 + **sizeof**(**int**), that is, with alignment constraints, **sizeof**(C) = 2 * **sizeof**(**int**).
However, it is consistent with the Standard to allow $i$ to overlap empty bases, which would lead to **sizeof**(C) = **sizeof**(**int**), which is exactly the result given by GNU G++. This optimization is called *empty base optimization*: empty bases can overlap the data of non-empty bases or fields.
Though, offsets for $B_1$ and $B_2$ are still distinct, as each has a subobject of type $A$: if they were not, then we would again have $a_1 = a_2$ in the program below:

```
C    c;
B1 * b1 = (B1 *) &c;
B2 * b2 = (B2 *) &c;
A  * a1 = (A  *) b1;
A  * a2 = (A  *) b2;
```

So, two empty bases may overlap only with care. We shall see later in more detail the precise constraints to follow when letting empty bases overlap other data. This is the main reason why C++ object layout cannot be mapped to C structure representations as Compcert [2] or cfront [8] do.
However, our work also shows that it is possible, without breaking the semantics of a C++ program, to generalize this empty base optimization to empty structure members.
This is the reason why our formalization gives no precise definition for empty classes. Instead, some constraints are enforced:

**Hypothesis 2.** *A layout algorithm may define its own notions of* empty *and* dynamic *classes, subject to the following constraints:*

- *An empty class must have no scalar fields*
- *An empty class must have no non-empty base*
- *An empty class must have no structure field of non-empty class type (regardless, however, of the number of array cells)*
- *A dynamic class must not be empty*

We shall see layout algorithms giving precise definitions for empty classes, sometimes more restrictive to lower compilation time, at the price of losing run-time optimizations on the programs being compiled.

### 3.1.4 Sizes, data sizes, and their non-virtual counterparts

Following the layout scheme above, we have to consider not only the size of a full object (which would be yielded by the sizeof operator), but also the size of the non-virtual parts of a class.
Moreover, dealing with empty classes that can overlap other components, considering their size or non-virtual size is not enough (as those sizes are necessary not null), we introduce the notion of *data size*. Roughly speaking, the *data size* of a class is an upper bound on the sum of the sizes of all scalar fields accessible from this class (through inheritance and/or structure array field paths). The *non-virtual data size* of a class is an upper bound on the sum of the sizes of all scalar fields accessible from this class *when the first inheritance step is non-virtual inheritance* (i.e. virtual inheritance is allowed only once a structure field within a non-virtual base of C is reached). Both sums also include the sizes of the dynamic type data for all relevant bases.
The data size and non-virtual data size are not relevant for empty classes.

### 3.1.5 Alignments and non-virtual alignments

Consider the following hierarchy:

```
struct A {
  double d0;
  char   c0;
};

struct B1: virtual A {
  int  i1;
  char c1;
};

struct B2: virtual A {
  int i2;
};

struct C: B1, B2 {
};
```

GNU GCC 4.3 on Intel x86 32-bit platform with option `-malign-double` (to align **double** on 8-byte boundaries) gives the following layouts within $C$:

- $B_1$ is laid out at offset 0
- $B_2$ is laid out at offset 12
- $A$ is laid out at offset 24

Even though $B_2$ inherits from $A$, it is laid out at offset 12 within $C$, which is not a multiple of 8 (the alignment of **double** `d0`). This shows that non-virtual bases $B_1$ and $B_2$ are aligned independently on the access to virtual base $A$. This leads us to distinguish two alignments for a class:

- the *alignment* of a class $C$, such that any instance of $C$ be aligned at an offset multiple of this alignment
- the *non-virtual alignment* of a class $C$, to allow the alignment of $C$ as a base of other classes, so that any subobject of type $C$ be aligned at an offset multiple of this alignment

In our case, we can see that the non-virtual alignment of $B_2$ is 4 instead of 8.

## 3.2 The expected output of a layout algorithm

**Definition 8.** *A* class layout *is a tuple of class-indexed families:*

$((\mathsf{dnvboff}_C), (\mathsf{vboff}_C), (\mathsf{foff}_C), (\mathsf{nvdsize}_C), (\mathsf{nvsize}_C), (\mathsf{dsize}_C), (\mathsf{size}_C), (\mathsf{nvalign}_C), (\mathsf{align}_C))_{C \in \mathcal{C}}$

*where, for each class $C$:*

- $\mathit{dnvboff}_C$ : $\mathcal{DNV}(C) \to \mathbb{N}$ *is a function assigning to each non-virtual direct base of* $C$ *an offset within a subobject of type* $C$*. This offset corresponds to the direct non-virtual subobject of type* $B$ *within a subobject of type* $C$*.*
- $\mathit{vboff}_C$ : $\mathcal{V}(C) \to \mathbb{N}$ *is a function assigning to each virtual base of* $C$ *an offset within a full instance of* $C$*.*
- $\mathit{foff}_C$ : $\mathcal{ScF}(C) \cup \mathcal{StF}(C) \to \mathbb{N}$ *is a function assigning to each (scalar or structure array) field of* $C$ *an offset within a subobject of type* $C$*.*
- $\mathit{nvdsize}_C$ *is the* non-virtual data size *of* $C$
- $\mathit{dsize}_C$ *is the* data size *of* $C$
- $\mathit{nvsize}_C$ *is the* non-virtual size *of* $C$
- $\mathit{size}_C$ *is the* size *of* $C$ *as would be given by the* sizeof *operator*
- $\mathit{nvalign}_C$ *is the* non-virtual alignment *of* $C$
- $\mathit{align}_C$ *is the* alignment *of* $C$

As the hierarchy is assumed to be well-founded, a class layout algorithm would be able to incrementally compute offsets for a given class $C$, assuming that for all classes $B \prec C$, offsets have already been computed.
Again, the Coq development does not use dependent types to represent a class layout. Instead, a class layout is a mapping $O \in \mathsf{PTree.t\ Offsets.t}$ where Offsets.t is a record type having fields such as $\mathsf{dnvboff}, \mathsf{vboff}, \mathsf{foff}, \ldots$. To enforce the domains, we externally constrain such mappings $o$ with *guard constraints*:

- To incrementally compute the layout of a class $C$, it is assumed that for all classes $B \prec C$, their layout data are defined:

$$\forall B \in \mathcal{C} : B \prec C \Rightarrow O(B) \neq \varnothing$$

- Conversely, we assume that all layout data are computed only for valid classes:

$$\forall B \in \mathbb{N}^* : O(B) \neq \varnothing \Rightarrow B \in \mathcal{C}$$

- For a class $B$ such that $O(B) = \{o\}$ for some layout data $o \in \mathsf{Offsets.t}$, the function $\mathsf{dnvboff}_B$ is represented by a mapping $o.\mathsf{dnvboff} \in \mathsf{PTree.t}\ \mathbb{N}^*$ instead of $\mathsf{PTree.t}\ \mathcal{DNV}(B)$ to avoid dependent types. So, the values have to be externally constrained:

$$\forall A \in \mathbb{N}^* : o.\mathsf{dnvboff}(A) \neq \varnothing \Leftrightarrow A \in \mathcal{DNV}(B)$$

We put those constraints along with the layout constraints of Section 3.3. Then, given such a class layout, it is possible to compute the offset of non-virtual paths within a subobject of some class $C$, and the offset of any inheritance path within a full instance of $C$.

**Definition 9.** *If $l$ is a non-virtual path from some class $C$ to some class $A$, then $\mathit{nvsoff}(l) \in \mathbb{N}$ shall denote its offset within a subobject of type $C$. It is computed as follows, by structural recursion on $l$:*

- $\mathit{nvsoff}(C :: \mathit{nil}) \underset{\text{def.}}{=\!=} 0$ *($C :: \mathit{nil}$ is the trivial non-virtual path to the subobject itself).*
- *if $l = C :: B :: l'$, then $B \in \mathcal{DNV}(C)$ and $B :: l'$ is a non-virtual path from $B$ to $A$, and $\mathit{nvsoff}(C :: B :: l') \underset{\text{def.}}{=\!=} \mathit{dnvboff}_C(B) + \mathit{nvsoff}(B :: l')$.*

In our Coq formalization, we represented $\mathsf{nvsoff}$ through a tail-recursive function $\mathsf{nvsoff}'$ with an additional accumulating argument:

$$\begin{aligned} \mathsf{nvsoff}'(accu, C :: \mathsf{nil}) &\underset{\text{def.}}{=\!=} accu \\ \mathsf{nvsoff}'(accu, C :: B :: l') &\underset{\text{def.}}{=\!=} \mathsf{nvsoff}'(accu + \mathsf{dnvboff}_C(B), B :: l') \end{aligned}$$

and we proved a rewriting lemma to show

$$\mathsf{nvsoff}'(accu_1, l) = \mathsf{nvsoff}'(accu_2, l) + accu_1 - accu_2$$

for any non-virtual path $l$ and any accumulators $accu_1, accu_2 \in \mathbb{Z}$. So:

$$\mathsf{nvsoff}(l) = \mathsf{nvsoff}'(0, l)$$

**Definition 10.** *If $(h, l)$ is a path from $C$ to $A$, then its offset $\mathit{soff}_C(h, l) \in \mathbb{N}$ within a full instance of $C$ is computed as follows:*

- $\mathit{soff}_C(\mathit{Repeated}, l) \underset{\text{def.}}{=\!=} \mathit{nvsoff}(l)$,
- $\mathit{soff}_C(\mathit{Shared}, l) \underset{\text{def.}}{=\!=} \mathit{vboff}_C(B) + \mathit{nvsoff}(l)$, *where $B \in \mathcal{V}(C)$ and $l$ is a non-virtual path from $B$ to $A$*

The above definition can be simplified by the convention $\mathsf{vboff}_C(C) \underset{\text{def.}}{=\!=} 0$, thanks to the well-foundedness of the hierarchy: indeed, $C$ is not a virtual base of itself, so $\mathsf{vboff}_C(C)$ is theoretically undefined. Then, with this convention, we obtain for any path $(h, B :: l')$ from $C$ to $A$:

$$\mathsf{soff}_C(h, B :: l') = \mathsf{vboff}_C(B) + \mathsf{nvsoff}(B :: l)$$

regardless of $h$. Remember that Lemma 7 (p. 12) makes $h$ depend on whether $B = C$ or not. This leads us to introduce the notion of *generalized virtual bases* of a class.

**Definition 11.** *The* generalized virtual bases *of $C$ are $C$ and the virtual bases of $C$. This set, denoted $\tilde{\mathcal{V}}(C)$, shall be the domain of $\mathsf{vboff}_C$:*

$$\tilde{\mathcal{V}}(C) \quad \underset{\text{def.}}{=\!=} \quad \mathcal{V}(C) \cup \{C\}$$

$$\begin{array}{rccl} \mathsf{vboff}_C: & \tilde{\mathcal{V}}(C) & \to & \mathbb{N} \\ & C & \mapsto & 0 \\ & B \in \mathcal{V}(C) & \mapsto & \mathsf{vboff}_C(B) \end{array}$$

**Lemma 8.** *If $l'$ is a non-virtual path from some class $B$ to some class $A$, then casting a subobject from $B$ to $A$ through path $l'$ is equivalent to adding the offset $\mathsf{nvsoff}(l')$ to the offset of the subobject. That is, for any path $P$ from $C$ to $B$:*

$$\mathsf{soff}_C(P@(\mathsf{Repeated}, l')) = \mathsf{soff}_C(P) + \mathsf{nvsoff}(l')$$

This shows that non-virtual derived-to-base casting does not depend on the starting subobject. However, to make a derived-to-base static cast through virtual inheritance, it is necessary to know the offset of the virtual base from the *full* object, so a subobject has to know its offset from the full object. See for instance Chen [3] who mechanically formalizes a type system for C++ multiple inheritance to determine which arithmetic operations are necessary for casts.

Given those definitions for inheritance path offsets, we can compute the offset of an array of $n'$ structures of type $C'$ within an array of $n$ structures of size $C$:

**Definition 12.** *Let $C, C'$ be two classes, and $n, n'$ two integers. If $l$ is an array path from $C[n]$ to $C'[n']$, then the offset $\mathsf{aoff}_C(l)$ of $C'[n']$ within $C[n]$ through the array path $l$ is computed as follows:*

- $$\mathsf{aoff}_C(\mathsf{nil}) \underset{\text{def.}}{=\!=} 0$$
- *if $l = (i, P, f) :: l'$, then $i$ is the index in the array $C[n]$, so $0 \leq i < n$, and there is a class $B$ such that $P$ is an inheritance path from $C$ to $B$, and $f$ is a field of $B$ declared as an array of $q$ structures of type $A$ for some $q$ and $A$, so that $l'$ is an array path from $A[q]$ to $C'[n']$. Then we define:*

$$\mathsf{aoff}_C((i, P, f) :: l') \underset{\text{def.}}{=\!=} i \cdot \mathsf{size}_C + \mathsf{soff}_C(P) + \mathsf{foff}_B(f) + \mathsf{aoff}_A(l')$$

**Definition 13** (Relative pointer)**.** *Let $p = (\mathcal{P}, i, P)$ be a generalized subobject (or relative pointer) from an array of some type $C[n]$, where $\mathcal{P}$ is an array path from $C[n]$ to $C'[n']$ for some class $C'$. Then, the offset of $p$ within $C[n]$ is denoted:*

$$\mathsf{off}(p) \underset{\text{def.}}{=\!=} \mathsf{aoff}_C(\mathcal{P}) + i \cdot \mathsf{size}_{C'} + \mathsf{soff}_{C'}(P)$$

### 3.3 Soundness

In our formalism, rather than imposing a fixed object layout as for C structures in Compcert [2], we propose some constraints on object layout, which layout algorithms must abide by. Then we prove that those constraints make two distinct scalar fields disjoint and two pointers to differents subobjects of the same static type point to different memory locations.

**Remark:** Constraint equations and inequations are numbered following the same scheme as in the POPL 2011 draft submission [6]. They may appear in a different order, though.

Consider a class $C$.

#### 3.3.1 Total size

The total size $\mathsf{size}_C$ of a class $C$ is considered as an upper bound on the sizes of all its components. In the same way, the total non-virtual size $\mathsf{nvsize}_C$ of a class $C$ is an upper bound on the sizes of all its non-virtual components (the non-virtual sizes of its direct non-virtual bases, and the sizes of its fields).

**Hypothesis 3.** *The non-virtual size of $C$ is an upper bound on the non-virtual sizes of all its direct non-virtual bases:*

$$\forall B \in \mathcal{DNV}(C) : \mathit{dnvboff}_C(B) + \mathit{nvsize}_B \leq \mathit{nvsize}_C \tag{C1}$$

**Lemma 9.** *If $l$ is a non-virtual path from $C$ to some class $A$, then the non-virtual part of $A$ is included in the non-virtual part of $C$:*

$$\mathit{nvsoff}(l) + \mathit{nvsize}_A \leq \mathit{nvsize}_C$$

*Proof.* By induction on the length of $l$.
If $l$ is the trivial path $C :: \mathsf{nil}$, then $A = C$, $\mathsf{nvsoff}(l) = 0$ and the inequality is an equality.
If $l = C :: B :: l'$, then $B \in \mathcal{DNV}(C)$ and $B :: l'$ is a non-virtual path from $B$ to $A$, and we have $\mathsf{nvsoff}(C :: B :: l') = \mathsf{dnvboff}_C(B) + \mathsf{nvsoff}(B :: l')$. By induction hypothesis, we have $\mathsf{nvsoff}(B :: l') + \mathsf{nvsize}_A \leq \mathsf{nvsize}_B$. By the above hypothesis, we have $\mathsf{dnvboff}_C(B) + \mathsf{nvsize}_B \leq \mathsf{nvsize}_C$, which concludes. □

**Hypothesis 4.** *The total size of $C$ is an upper bound on the non-virtual sizes of all its generalized virtual bases (including $C$ itself):*

$$\forall B \in \tilde{\mathcal{V}}(C) : \mathit{vboff}_C(B) + \mathit{nvsize}_B \leq \mathit{size}_C \tag{C3}$$

**Lemma 10.** *If $(h, l)$ is an inheritance path (= base class subobject) from $C$ to some class $A$, then the non-virtual part of $A$ is included in $C$:*

$$\mathit{soff}_C(h, l) + \mathit{nvsize}_A \leq \mathit{size}_C$$

*Proof.* Recall that $l = B :: l'$ for some class $B \in \tilde{\mathcal{V}}(C)$ and some $l'$ such that $B :: l'$ is a non-virtual path from $B$ to $A$. We then have $\mathsf{soff}_C(h, l) = \mathsf{vboff}_C(B) + \mathsf{nvsoff}(B :: l')$. (C3, p. 20) gives us $\mathsf{vboff}_C(B) + \mathsf{nvsize}_B \leq \mathsf{size}_C$. Lemma 9 (p. 19) gives us $\mathsf{nvsoff}(B :: l') + \mathsf{nvsize}_A \leq \mathsf{nvsize}_B$, which concludes. □

The non-virtual part of a class $C$ contains all fields defined in $C$. To express this condition, we must define the *size* of a field.

**Notation 4.** *We assume that there exists a function* $\mathsf{scsize} : \mathcal{A} \cup \mathcal{C} \to \mathbb{N}^*$ *computing the size of a scalar value (atom, or pointer to* $C$ *for some class* $C$*).*

The function $\mathsf{scsize}$ should not depend on the layout algorithm. For most compilers, the size of a value of type pointer to $T$ does not depend on $T$. But such assumption is not relevant in our formalization.

**Definition 14.** *The* size $\mathit{fsize}(f)$ *of a field* $f$ *is defined as follows:*

- $\mathit{fsize}(f, t) \underset{\text{def.}}{=\!=} \mathit{scsize}(t)$ *for a scalar field*
- $\mathit{fsize}(f, B, n) \underset{\text{def.}}{=\!=} n \cdot \mathit{size}_B$ *for a structure array field*

**Hypothesis 5.** *The total non-virtual size of* $C$ *is an upper bound on the sizes of all its fields:*

$$\forall F \in \mathcal{F}(C) : \mathit{foff}_C(F) + \mathit{fsize}(F) \leq \mathit{nvsize}_C \tag{C2}$$

**Lemma 11.** *If* $l$ *is an array path from* $C[n]$ *to* $C'[n']$*, then the designated array* $C'[n']$ *is "included" in* $C[n]$*:*

$$\mathit{aoff}_C(l) + n' \cdot \mathit{size}_{C'} \leq n \cdot \mathit{size}_C$$

*Proof.* By structural induction on $l$.

- If $l = \mathsf{nil}$, then $C' = C$ and $n' = n$ and $\mathsf{aoff}_C(l) = 0$, so the inequality is an equality.
- Otherwise, $l = (i, (h, p), f) :: l'$ where $0 \leq i < n$, $(h, p)$ is an inheritance path from $C$ to some class $B$, and $f$ is a structure array of some type $A[m]$ defined in $B$. So, we have $\mathsf{aoff}_C(l) = i \cdot \mathsf{size}_C + \mathsf{soff}_C(h, p) + \mathsf{foff}_B(f) + \mathsf{aoff}_A(l')$, and :
  - by induction hypothesis, $\mathsf{aoff}_A(l') + n' \cdot \mathsf{size}_{C'} \leq m \cdot \mathsf{size}_A = \mathsf{fsize}(f)$ by definition
  - $\mathsf{foff}_B(f) + \mathsf{fsize}(f) \leq \mathsf{nvsize}_B$ by (C2, p. 20)
  - $\mathsf{soff}_C(h, p) + \mathsf{nvsize}_B \leq \mathsf{size}_C$ by Lemma 10 (p. 20)
  - $i + 1 \leq n$ by hypothesis

which concludes.

□

**Corollary 12.** *If $p$ is a relative pointer (= generalized subobject) from $C[n]$ to some class $A$, then the non-virtual part of $A$ is included in $C[n]$:*

$$\mathit{off}_C(p) + \mathit{nvsize}_A \leq n \cdot \mathit{size}_C$$

*Proof.* Immediately follows from Lemmata 10 (p. 20) and 11 (p. 21). □

### 3.3.2 Alignment

The non-virtual alignment of a class $C$ is a boundary used to align $C$ as a non-virtual base of another class. So, it has to be a multiple of the alignments of all the non-virtual components of $C$: the alignments of fields defined in $C$, and the non-virtual alignments of non-virtual bases of $C$.
So we must first define field alignments.

**Notation 5.** *We assume that there exists a function $\mathsf{scalign} : \mathcal{A} \cup \mathcal{C} \to \mathbb{N}^*$ computing the alignment of a scalar value (atom, or pointer to $B$ for some class $B$).*

As with scsize, the function scalign should not depend on the layout algorithm.

**Definition 15.** *The* alignment $\mathit{falign}(f)$ *of a field $f$ is defined as follows:*

- $\mathit{falign}(f, t) \underset{\text{def.}}{=\!=} \mathit{scalign}(t)$ *for a scalar field*
- $\mathit{falign}(f, B, n) \underset{\text{def.}}{=\!=} \mathit{align}_B$ *for a structure array field*

Indeed, a structure array field contains "full" instances of some class $B$, so the field must be aligned to the whole alignment of $B$, not only the non-virtual alignment of $B$.

**Hypothesis 6.** *The non-virtual alignment of a class $C$ is a multiple of the alignments of its fields:*

$$\forall F \in \mathcal{F}(C) : (\mathit{falign}(F) \mid \mathit{nvalign}_C) \tag{C14b}$$

**Hypothesis 7.** *Any field $f$ of $C$ must be laid out with respect to its alignment:*

$$\forall F \in \mathcal{F}(C) : (\mathit{falign}(F) \mid \mathit{foff}_C(f)) \tag{C14a}$$

**Hypothesis 8.** *The non-virtual alignment of a class $C$ is a multiple of the non-virtual alignments of its direct non-virtual bases:*

$$\forall B \in \mathcal{DNV}(C) : (\mathit{nvalign}_B \mid \mathit{nvalign}_C) \tag{C15b}$$

**Hypothesis 9.** *Any non-virtual base $B$ of $C$ must be laid out with respect to its non-virtual alignment:*

$$\forall B \in \mathcal{DNV}(C) : (\mathit{nvalign}_B \mid \mathit{dnvboff}_C(B)) \tag{C15a}$$

Finally, the alignment of a class $C$ synthesizes the non-virtual alignments of $C$ and its virtual bases.

**Hypothesis 10.** *The alignment of a class $C$ is a multiple of the non-virtual alignments of its generalized virtual bases (including $C$ itself):*

$$\forall B \in \tilde{\mathcal{V}}(C) : (\mathit{nvalign}_B \mid \mathit{align}_C) \tag{C17b}$$

**Hypothesis 11.** *Any virtual base $B$ of $C$ must be laid out with respect to its non-virtual alignment:*

$$\forall B \in \mathcal{V}(C) : (\mathit{nvalign}_B \mid \mathit{vboff}_C(B)) \tag{C17a}$$

(C17a, p. 22) is also true for $B = C$, as $\mathsf{vboff}_C(C) \underset{\text{def.}}{=\!=} 0$.
Then, all those constraints allow to show that the access to a field is correctly aligned.

**Lemma 13.** *If $l$ is a non-virtual path from some class $C$ to some class $A$, then the access to $A$ through $l$ is correctly aligned:*

$$(\mathit{nvalign}_A \mid \mathit{nvalign}_C) \tag{$i$}$$

$$(\mathit{nvalign}_A \mid \mathit{nvsoff}(l)) \tag{$ii$}$$

*Proof.* By structural induction on $l$.

- If $l = \mathsf{nil}$, then $C = A$ and $\mathsf{nvsoff}(l) = 0$, which trivially concludes.
- Otherwise, $l = B :: l'$ with $B \in \mathcal{DNV}(C)$ and $\mathsf{nvsoff}(l) = \mathsf{dnvboff}_C(B) + \mathsf{nvsoff}(l')$. So:
  - $(\mathsf{nvalign}_B \mid \mathsf{nvalign}_C)$ by (C14a, p. 22) and $(\mathsf{nvalign}_A \mid \mathsf{nvalign}_B)$ by induction hypothesis, thus $(i)$ by transitivity.
  - $(\mathsf{nvalign}_B \mid \mathsf{dnvboff}_C(B))$ by (C14b, p. 22), and $(\mathsf{nvalign}_A \mid \mathsf{nvalign}_B)$ by induction hypothesis, and $(\mathsf{nvalign}_A \mid \mathsf{nvsoff}(l'))$ by induction hypothesis, thus $(ii)$ by transitivity and compatibility of $(. \mid .)$ with addition.

□

**Lemma 14.** *If $(h, l)$ is an inheritance path (= base class subobject) from some class $C$ to some class $A$, then the access to $A$ through $(h, l)$ is correctly aligned:*

$$(\mathit{nvalign}_A \mid \mathit{align}_C) \tag{$i$}$$

$$(\mathit{nvalign}_A \mid \mathit{soff}_C(h, l)) \tag{$ii$}$$

*Proof.* Recall that $l = B :: l'$ for some class $B \in \tilde{\mathcal{V}}(C)$ and some $l'$ such that $B :: l'$ is a non-virtual path from $B$ to $A$. We then have $\mathsf{soff}_C(h, l) = \mathsf{vboff}_C(B) + \mathsf{nvsoff}(B :: l')$.

- (C17a, p. 22) gives us $(\mathsf{nvalign}_B \mid \mathsf{align}_C)$; part $(i)$ of Lemma 13 (p. 22) gives us $(\mathsf{nvalign}_A \mid \mathsf{nvalign}_B)$, thus $(i)$ by transitivity.
- (C17b, p. 22) gives us $(\mathsf{nvalign}_B \mid \mathsf{vboff}_C(B))$; Lemma 13 (p. 22) gives us $(\mathsf{nvalign}_A \mid \mathsf{nvalign}_B)$ and $(\mathsf{nvalign}_A \mid \mathsf{nvsoff}(l'))$, thus $(ii)$ by transitivity and compatibility of $(. \mid .)$ with addition.

□

If $C$ is a class and $c$ is an array of several structures of type $C$, then access to the second structure item of $c$ will be realized at low-level through adding an offset of **sizeof**(C) bytes to a pointer to $c$. So, the following constraint is necessary to ensure the correct alignment of such an access:

**Hypothesis 12.**

$$(\mathit{align}_C \mid \mathit{size}_C) \tag{C18}$$

This condition will ensure the:

**Lemma 15.** *If $l$ is an array path from $C[n]$ to $C'[n']$, then the designated array $C'[n']$ is correctly aligned in $C[n]$:*

$$(\mathit{align}_{C'} \mid \mathit{align}_C) \tag{$i$}$$

$$(\mathit{align}_{C'} \mid \mathit{aoff}_C(l)) \tag{$ii$}$$

*Proof.* By structural induction on $l$.

- If $l = \mathsf{nil}$, then $C' = C$ and $\mathsf{aoff}_C(l) = 0$, which trivially concludes.
- Otherwise, $l = (i, (h, p), f) :: l'$ where $0 \leq i < n$, $(h, p)$ is an inheritance path from $C$ to some class $B$, and $f$ is a structure array of some type $A[m]$ defined in $B$, such that $\mathsf{falign}(f) = \mathsf{align}_A$. So, we have $\mathsf{aoff}_C(l) = i \cdot \mathsf{size}_C + \mathsf{soff}_C(h, p) + \mathsf{foff}_B(f)$, and:
  - $(\mathsf{align}_{C'} \mid \mathsf{align}_A)$ by induction hypothesis, and $(\underbrace{\mathsf{falign}(f)}_{=\,\mathsf{align}_A} \mid \mathsf{nvalign}_B)$ by (C15b, p. 22), and $(\mathsf{nvalign}_B \mid \mathsf{align}_C)$ by part $(i)$ of Lemma 14 (p. 23), thus $(i)$ by transitivity
  - thanks to $(i)$ and (C18, p. 23), we have $(\mathsf{align}_{C'} \mid i \cdot \mathsf{size}_C)$. Moreover, by induction hypothesis, $(\mathsf{align}_{C'} \mid \mathsf{align}_A)$, and $(\mathsf{align}_A \mid \mathsf{nvalign}_B)$ as above, and $(\mathsf{nvalign}_B \mid \mathsf{soff}_C(h, p))$ by part $(ii)$ of Lemma 14 (p. 23), and $(\mathsf{align}_A \mid \mathsf{foff}_B(f)$ by (C15a, p. 22), and $(\mathsf{align}'_C \mid \mathsf{aoff}_A(l')$ by induction hypothesis, thus $(ii)$.

□

**Corollary 16.** *If $p$ is a relative pointer (= generalized subobject) from $C[n]$ to some class $A$, then the access to $A$ through $p$ is correctly aligned:*

$$(\mathit{nvalign}_A \mid \mathit{align}_C)$$

$$(\mathit{nvalign}_A \mid \mathit{off}_C(p))$$

*Proof.* Immediately follows from Lemmata 14 (p. 23) and 15 (p. 24). □

This lemma stresses out the difference of use between the alignment and the non-virtual alignment of a class: whereas full instances are aligned to the full alignment of their class, by contrast, generalized subobjects are aligned to the non-virtual alignment of their static type.
This finally allows to show the:

**Theorem 1.** *If $p$ is a generalized subobject of static type $A$ from a structure array of $C$, and if $(f,t)$ is a scalar field of $A$ of scalar type $t \in \mathcal{A} \cup \mathcal{C}$, then the access to this field is correctly aligned:*

$$(\mathit{scalign}(t) \mid \mathit{align}_C)$$

$$(\mathit{scalign}(t) \mid \mathit{off}_C(p))$$

*Proof.* Immediately follows from Corollary 16 (p. 24) and conditions (C15a, p. 22) and (C15b, p. 22). □

### 3.3.3 Data size

We aim at stating constraints to forbid the overlapping of different scalar data. So, we shall only consider here *non-empty* components, in the sense that roughly speaking, a component is non-empty as soon as it contains some scalar data.
Then, to express that different scalar data do not overlap, we shall not use the sizes, but the *data sizes* of the relevant components. In this section, we shall show that data sizes are not relevant for empty components.
Recall that the actual definition for an empty class is left to the layout algorithm. Based on this definition, we may introduce the notion of *empty field*, which is a structure field of an empty class type, regardless of the number of its array elements:

**Notation 6.** *The set of* empty fields *of $C$ is the set of those fields defined in class $C$ that are array structures of type $B$ for some empty class $B$:*

$$E\mathcal{F}(C) \underset{\text{def.}}{=\!=} \{(f,B,n) \in St\mathcal{F}(C) \mid B \text{ is empty}\}$$

*The set of non-empty fields of $C$ is:*

$$NE\mathcal{F}(C) \underset{\text{def.}}{=\!=} Sc\mathcal{F}(C) \cup St\mathcal{F}(C) \backslash E\mathcal{F}(C)$$

Then, we may define the notion of the *data size* of a field. For a scalar field, the data size of a field is equal to its size. But for a structure array field, there are two cases:

- if the field has only one structure element of some type $B$, then the data size of such a field is equal to the data size of the class $B$.
- otherwise, it is important to note that the data of a field is considered to be a *contiguous interval*. So, any padding between two array elements is lost. However, nothing prevents us from reusing the padding of the last element of the array.

For this reason, we define the *data size* of a non-empty field as follows.

**Definition 16.** *The* data size *of a scalar field* $(f, t)$ *is:*

$$\mathit{fdsize}(f, t) \underset{\text{def.}}{=\!=} \mathit{scsize}(t)$$

*The* data size *of a non-empty structure array field* $(f, b, n)$ *is:*

$$\mathit{fdsize}(f, b, n) \underset{\text{def.}}{=\!=} (n-1) \cdot \mathit{size}_B + \mathit{dsize}_B$$

Then, in the same way as for the whole sizes, we introduce the notions of *non-virtual data size* (for the non-virtual part of a class) and *data size* (for the whole class), as bounds on the sizes of the class components.

**Notation 7.** *In the same way as for fields, we shall use the following notations:*

- $NE\mathcal{DNV}(C)$ *for the set of non-empty direct non-virtual bases of* $C$
- $E\mathcal{DNV}(C)$ *for the set of empty direct non-virtual bases of* $C$
- $NE\mathcal{V}(C)$ *for the set of non-empty virtual bases of* $C$
- $E\mathcal{V}(C)$ *for the set of empty virtual bases of* $C$
- $NE\tilde{\mathcal{V}}(C)$ *for the set of non-empty generalized virtual bases of* $C$
- $E\tilde{\mathcal{V}}(C)$ *for the set of empty generalized virtual bases of* $C$

Then, the non-virtual data of a class $C$ is divided into two parts, each being a *contiguous interval*:

- the non-virtual data of the non-empty direct non-virtual bases of $C$
- the data of the fields defined in $C$

**Hypothesis 13.** *There exists a boundary* $\mathsf{fboundary}_C \in \mathbb{N}$ *between the non-virtual data for the direct non-virtual bases of* $C$ *and the* non-empty *fields of* $C$*:*

$$\forall B \in NE\mathcal{DNV}(C) : \mathsf{dnvboff}_C(B) + \mathsf{nvdsize}_B \leq \mathsf{fboundary}_C \tag{C7}$$

$$\forall F \in NE\mathcal{F}(C) : \mathsf{fboundary}_C \leq \mathsf{foff}_C(F) \tag{C8}$$

That hypothesis ensures that the two parts are disjoint.
In our formalization, such boundary is included in the definition of a class layout (i.e. $\mathsf{fboundary}_C$ is a value explicitly computed by the layout algorithm).

**Hypothesis 14.** *The* non-virtual data size *of class* $C$ *is an upper bound of the sizes of the two parts of the non-virtual data of* $C$*:*

$$\mathsf{fboundary}_C \leq \mathsf{nvdsize}_C \tag{C11}$$

$$\forall F \in NE\mathcal{F}(C) : \mathsf{foff}_C(F) + \mathsf{fdsize}(F) \leq \mathsf{nvdsize}_C \tag{C10}$$

(C11, p. 26) is necessary if there are no non-empty fields defined in $C$. Otherwise, (C11, p. 26) is entailed by (C10, p. 26).

**Lemma 17.** *If* $l$ *is a non-virtual path from* $C$ *to some* non-empty *class* $A$*, then the non-virtual part of* $A$ *is included in the non-virtual part of* $C$*:*

$$\mathsf{nvsoff}(l) + \mathsf{nvdsize}_A \leq \mathsf{nvdsize}_C$$

*Proof.* Same shape as the proof for Lemma 9 (p. 19), additionally using (C11, p. 26). Additionally to $A$, all considered classes are not empty, as they have a non-empty base $A$. □

**Hypothesis 15.** *The total data size of* $C$ *is an upper bound on the non-virtual data sizes of all its non-empty generalized virtual bases (including* $C$ *itself, if non-empty):*

$$\forall B \in NE\tilde{\mathcal{V}}(C) : \mathsf{vboff}_C(B) + \mathsf{nvdsize}_B \leq \mathsf{dsize}_C \tag{C13}$$

**Lemma 18.** *If* $(h, l)$ *is an inheritance path (= base class subobject) from* $C$ *to some class* $A$*, then the non-virtual data of* $A$ *is included in* $C$*:*

$$\mathsf{soff}_C(h, l) + \mathsf{nvdsize}_A \leq \mathsf{dsize}_C$$

*Proof.* Same shape as the proof for Lemma 10 (p. 20). □

Recall that the *data* of a structure array of $n$ cells of type $C$ consisting of a *contiguous interval* embedding the total sizes of the first $n-1$ cells and the only data of the last cell:

$$(n-1) \cdot \mathsf{size}_C + \mathsf{dsize}_C$$

**Lemma 19.** *If $p = (l, i, P)$ is a relative pointer (= generalized subobject) from $C[n]$ to some non-empty class $A$, then the non-virtual data of $A$ is included in the data of $C[n]$:*

$$\mathit{off}_C(p) + \mathit{nvdsize}_A \leq (n-1) \cdot \mathit{size}_C + \mathit{dsize}_C$$

*Proof.* By structural induction on $l$.

- If $l = \mathsf{nil}$, then $i \leq (n-1)$ and $\mathsf{off}_C(p) = i \cdot \mathsf{size}_C + \mathsf{soff}_C(P)$; moreover, $P$ is an inheritance path from $C$ to $A$, so by Lemma 18 (p. 27), $\mathsf{soff}_C(P) + \mathsf{nvdsize}_A \leq \mathsf{dsize}_C$, which concludes.
- Otherwise, $l = (i', (h', p'), f') :: l'$ where $0 \leq i' < n$, $(h', p')$ is an inheritance path from $C$ to some class $B$, and $f'$ is a structure array of some type $C'[n']$ defined in $B$, such that $(l', i, P)$ is a relative pointer from $C'[n']$ to $A$, and $\mathsf{off}_C(p) = i \cdot \mathsf{size}_C + \mathsf{soff}_C(h, p) + \mathsf{foff}_B(f') + \mathsf{off}_{C'}(l', i, P)$, and :
  - by induction hypothesis, $\mathsf{off}_{C'}(l', i, P) + \mathsf{nvdsize}_A \leq (n'-1) \cdot \mathsf{size}_{C'} + \mathsf{dsize}_{C'} = \mathsf{fdsize}(f')$ by definition
  - $\mathsf{foff}_B(f) + \mathsf{fdsize}(f) \leq \mathsf{nvdsize}_B$ by (C11, p. 26) (legal because, as $A$ is not empty, neither is $B$)
  - $\mathsf{soff}_C(h, p) + \mathsf{nvdsize}_B \leq \mathsf{dsize}_C$ by Lemma 18 (p. 27)
  - $i' \leq n - 1$ by hypothesis

  which concludes.

□

### 3.3.4 Non-overlapping of data

To express that two fields $F_1$ and $F_2$ defined in some class $C$ should not overlap, we could expect their full size intervals to be disjoint:

$$[\mathsf{foff}_C(F_1), \mathsf{foff}_C(F_1) + \mathsf{fsize}(F_1)) \perp [\mathsf{foff}_C(F_2), \mathsf{foff}_C(F_2) + \mathsf{fsize}(F_2))$$

This condition is actually enforced by the Itanium C++ ABI, as seen in Section 4.1. However, it impedes the reusing of tail padding, even alignment tail padding. For instance, consider the following structure:

```
struct Z {};
struct A: Z {
  int  i;
  char ca;
};
struct B {
  A a;
  char cb;
}
```

Most compilers (such as GCC) lay out $a$ and $cb$ completely disjointly within $B$, laying out $cb$ at offset 8 (on Intel x86 32-bit platforms). This incurs the loss of unused space between the end of $a.ca$ and the beginning of $cb$: 3 bytes for alignment padding.
Our formalization allows reusing this padding, by defining *data sizes* distinct from sizes, trying to exclude padding as most as possible. Then, we introduce constraints to ensure that *data* intervals be disjoint.
In our example, whereas the size of $a$ is 8, its data size would be only 5 thanks to our formalization, such that $cb$ would be laid out at offset 5 instead of 8. Thus, we have to use the notion of *data size* rather than size, to express that two fields of the same class do not overlap:

**Hypothesis 16.** *The data of two non-empty fields defined in the same class do not overlap:*
$\forall F_1, F_2 \in NE\mathcal{F}(C)$ :

$$\begin{array}{rl} & [\mathit{foff}_C(F_1), \mathit{foff}_C(F_1) + \mathit{fdsize}(F_1)) \\ \perp & [\mathit{foff}_C(F_2), \mathit{foff}_C(F_2) + \mathit{fdsize}(F_2)) \end{array} \tag{C9}$$

Similarly, the non-virtual data of two direct non-virtual bases of $C$ do not overlap:

**Hypothesis 17.** $\forall B_1, B_2 \in \mathcal{DNV}(C)$ :

$$\begin{array}{rl} & [\mathit{dnvboff}_C(B_1), \mathit{dnvboff}_C(B_1) + \mathit{nvdsize}_{B_1}) \\ \perp & [\mathit{dnvboff}_C(B_2), \mathit{dnvboff}_C(B_2) + \mathit{nvdsize}_{B_2}) \end{array} \tag{C6}$$

However, non-virtual sizes may overlap. Consider for instance the following hierarchy:

```
struct A       {};
struct B       {int i};
struct C: A, B {};
```

Then, the following layout (actually given by GNU G++) complies to the above constraints:

$$\begin{array}{rcl} \mathsf{nvdsize}_A & = & 0 \\ \mathsf{nvsize}_A & = & 1 \\ \mathsf{foff}_B(i, \mathsf{int}) & = & 0 \\ \mathsf{nvdsize}_B & = & \mathsf{scsize(int)} \\ \mathsf{nvsize}_B & = & \mathsf{scsize(int)} \\ \mathsf{dnvboff}_C(A) & = & 0 \\ \mathsf{dnvboff}_C(B) & = & 0 \\ \mathsf{nvdsize}_C & = & \mathsf{scsize(int)} \\ \mathsf{nvsize}_C & = & \mathsf{scsize(int)} \end{array}$$

Here, the non-virtual sizes of $A$ and $B$ overlap in $C$, but this does not impede the access to field $i$ of $B$, as $A$ is empty.
The actual purpose of distinguishing sizes from data sizes is that a pointer to an empty class can point outside of the actual data (accessible non-empty fields), but must still point inside the whole size of the object (so as to prevent it from pointing into another object that would be created independently). However, we shall see further in more detail tighter conditions to prevent two pointers to different subobjects of static type $A$ (where $A$ is an empty class) from pointing to the same memory location.

**Lemma 20.** *If $l_1, l_2$ are two distinct non-virtual paths from some class $C$ to some non-empty classes $B_1, B_2$, then their field data zones are disjoint:*

$$\bot \quad \begin{matrix} \left[\mathit{nvsoff}(l_1) + \mathit{fboundary}_{B_1}, \mathit{nvsoff}(l_1) + \mathit{nvdsize}_{B_1}\right) \\ \left[\mathit{nvsoff}(l_2) + \mathit{fboundary}_{B_2}, \mathit{nvsoff}(l_2) + \mathit{nvdsize}_{B_2}\right) \end{matrix}$$

*Proof.* For symmetry reasons, we may assume $\mathsf{length}(l_1) \leq \mathsf{length}(l_2)$. Then, there are two cases:

- either there is a non-virtual non-trivial path $B_1 :: A :: l'$ from $B_1$ to $B_2$ such that $l_2 = l_1 @_{\mathsf{Repeated}}(B_1 :: A :: l')$. So $A$ is a non-virtual direct base of $B_1$, and actually the non-virtual data of $A$ (which includes the non-virtual data of $B_2$, in particular field $F_2$) is disjoint from the field data of $B_1$ (in particular field $F_1$) thanks to the boundary $\mathsf{fboundary}_{B_1}$.
- or there are a class $A$ and two distinct non-virtual direct bases $A_1$ and $A_2$ of $A$, and a list $L$ such that for each $i \in \{1, 2\}$, $L{+}{+}A :: A_i :: \mathsf{nil}$ is a non-virtual path from $C$ to $A_i$ and $A_i :: l'_i$ is a non-virtual path from $A_i$ to $B_i$ for some $l'_i$. The non-virtual direct bases $A_1$ and $A_2$ of $A$ have disjoint non-virtual data. The non-virtual data of $A_i$ includes the non-virtual data of $B_i$, and in particular field $F_i$, so those fields are disjoint.

□

Recall the convention $\mathsf{vboff}_C(C) \underset{\text{def.}}{=\!=} 0$. This allows to roughly say that the non-virtual data of all non-empty generalized virtual bases of $C$ (with $C$ considered as a generalized virtual base of itself) are laid out one after another, in such a way that they are disjoint:

**Hypothesis 18.** $\forall B_1, B_2 \in NE\tilde{\mathcal{V}}(C)$ : *if $B_1, B_2$:*

$$\bot \quad \begin{matrix} [\mathit{vboff}_C(B_1), \mathit{vboff}_C(B_1) + \mathit{nvdsize}_{B_1}) \\ [\mathit{vboff}_C(B_2), \mathit{vboff}_C(B_2) + \mathit{nvdsize}_{B_2}) \end{matrix} \tag{C12}$$

It follows immediately that:

**Lemma 21.** *If $P_1, P_2$ are distinct inheritance paths from $C$ to some $B_1, B_2$, then their field data zones are disjoint:*

$$\left[\mathsf{soff}_C(P_1) + \mathsf{fboundary}_{B_1}, \mathsf{soff}_C(P_1) + \mathsf{nvdsize}_{B_1}\right) \perp \left[\mathsf{soff}_C(P_2) + \mathsf{fboundary}_{B_2}, \mathsf{soff}_C(P_2) + \mathsf{nvdsize}_{B_2}\right)$$

*Proof.* Let $P_i = (h_i, A_i :: l_i)$ for each $i$. If $A_1 = A_2$, then Lemma 20 (p. 29) about non-virtual paths can be reused. Otherwise, we know by (C12, p. 30) that the non-virtual data of $A_i$ are disjoint, then Lemma 17 (p. 27) and (C11, p. 26) conclude. □

The conditions given so far are enough to show that two different scalar fields reachable from two base class subobjects of an object are disjoint. However, when it comes to traversing structure array fields, we must show that two fields reachable from two different cells of the same array are disjoint. So far there is no condition ensuring such a property. Indeed, there is yet no link between the data size and the size of a class: we have to explicitly constrain that:

**Hypothesis 19.** *For any class $C$:*

$$\mathsf{dsize}_C \leq \mathsf{size}_C \tag{C4}$$

**Notation 8.** *The* length *of a pointer $p = (\mathcal{P}, j, P)$, denoted $\mathsf{plength}(p)$, is the length of its array path $\mathcal{P}$:*

$$\mathsf{plength}(p) \underset{\text{def.}}{=\!=} \mathsf{length}(\mathcal{P})$$

**Theorem 2.** *If $p_i$ are two generalized subobjects of static type $B_i$ within a structure array of type $C$, and if $F_i$ are two* ***scalar*** *fields defined in class $B_i$, such that $(p_1, F_1) \neq (p_2, F_2)$, then those two fields are disjoint.*

*Proof.* For symmetry reasons, we may assume $\mathsf{plength}(p_1) \leq \mathsf{plength}(p_2)$. Reason by induction on $\mathsf{plength}(p_1)$. Then, we shall introduce an alternate representation for $p_i$:

**Definition 17** (Alternate representation of generalized subobjects (= alternate relative pointers))**.** *For any generalized subobject (= relative pointer) $p = (\mathcal{P}, j, P)$ from a structure array of type $C$ to some class $B$, we can find $j' \in \mathbb{Z}$, an inheritance path $P'$ from $C$ to some class $B'$, and an* option *$\Phi$ such that:*

- *either $\Phi = \varnothing$ and $p = (\mathsf{nil}, j', P')$*
- *or there is a structure array field $F'$ and a relative pointer $p' =$ for some array path $\mathcal{P}'$ such that $\Phi = \{(F', (\mathcal{P}', j, P))\}$ and $\mathcal{P} = (j', P', F') :: \mathcal{P}'$ (so that $\mathsf{plength}(\mathcal{P}', j, P) = \mathsf{plength}(p) - 1$ to allow induction).*

$(j', P', \Phi)$ *is called the* alternate representation *of* $p$*, and we note:*

$$(\mathcal{P}, j, P) \propto (j', P', \Phi)$$

**Lemma 22.** *The alternate representation of a relative pointer* $p$ *is unique:*

$$p \propto \mathfrak{p}_1 \wedge p \propto \mathfrak{p}_2 \;\Rightarrow\; \mathfrak{p}_1 = \mathfrak{p}_2$$

This alternate representation is intended for proofs about concrete object layout, by contrast to the "regular" Definition 6 (p. 9) which is designed for the abstract high-level semantics of C++ (especially to model casts and other dynamic operations such as virtual method dispatch).
This alternate representation is trivially compatible with offsets:

**Lemma 23.** *Let* $p \propto (j', P', \Phi)$ *be a relative pointer from* $C$ *of static type* $B$*.*

- *if* $\Phi = \varnothing$*, then:*
$$\mathit{off}_C(p) = j' \cdot \mathit{size}_C + \mathit{soff}_C(P')$$
- *otherwise,* $P'$ *is an inheritance path from* $C$ *to some class* $A$ *and* $\Phi = \{(F', p')\}$ *for some structure field* $F'$ *of type* $B'$ *defined in* $A$ *and for some relative pointer* $p'$ *from* $B$ *of static type* $B$*, and:*
$$\mathit{off}_C(p) = j' \cdot \mathit{size}_C + \mathit{soff}_C(P') + \mathit{foff}_A(F') + \mathit{off}_{B'}(p')$$

In particular, the following interesting lemma holds:

**Lemma 24.** *If* $p \propto (j', P', \Phi)$ *is a relative pointer from* $C$ *of static type* $B$*, such that* $P'$ *is an inheritance path from* $C$ *to some class* $C'$*, then the field data of* $p$ *is included in the field data of* $C'$*:*

$$\begin{array}{ll} & [\mathit{off}_C(p) + \mathit{fboundary}_B, \mathit{off}_C(p) + \mathit{nvdsize}_B) \\ \subseteq & [j' \cdot \mathit{size}_C + \mathit{soff}_C(P') + \mathit{fboundary}_{C'}, j' \cdot \mathit{size}_C + \mathit{soff}_C(P') + \mathit{nvdsize}_{C'}) \end{array}$$

Then, for each $i$, let $p_i \propto (j'_i, P'_i, \Phi_i)$. There are several cases:

- if $j'_1 \neq j'_2$, then we have two disjoint cells of an array of structures of some type $C$. Thanks to (C4, p. 31), their data are also disjoint, which concludes.
- $j'_1 = j'_2$ and $P'_1 \neq P'_2$: then, thanks to the above lemma, the theorem for virtual inheritance directly applies.
- Now assume $j'_1 = j'_2$, $P'_1 = P'_2$.
  - If $\Phi_1 = \Phi_2$, then $F_1 \neq F_2$ are distinct fields of the same subobject, so by (C9, p. 28) they are disjoint.

- Otherwise, we can assume $\Phi_2 = \{F'_2, p'_2\}$ (because $\mathsf{plength}(p_1) \leq \mathsf{plength}(p_2)$, so that if $\Phi_2 = \varnothing$, then $\Phi_1 = \varnothing$).
  - if $\Phi_1 = \varnothing$, then $F_1$ and $F'_2$ are distinct fields (because $F_1$ is scalar whereas $F'_2$ is a structure field) of the same class, so their data are disjoint.
  - Otherwise, $\Phi_1 = \{F'_1, p'_1\}$.
    - if $F'_1 \neq F'_2$, then they are two distinct fields of the same class, so their data are disjoint.
    - Otherwise, we may use the induction hypothesis.

□

Our layout constraints allow to produce smarter layouts than GNU G++. Indeed, fields are laid out not by their whole sizes, but only by their data sizes. This allows fields to be stored within the end of a structure array field. Consider for instance:

```
struct A                    {};
struct B1: A                {};
struct B2: A                {};
struct C:  B1, B2           { char i; };
struct D                    { C    c; char j; };
```

Assuming $\mathsf{sizeof}(\mathsf{char}) = 1$, GNU G++ gives $\mathsf{sizeof}(\mathsf{C}) = 2$, as 2 different offsets to subobjects of type $A$ have to be allotted.
But GNU G++ would give $\mathsf{sizeof}(\mathsf{D}) = 3$ by laying out $j$ beyond the size of field $c$, so at offset 2, whereas our formalization allows $j$ to be laid out at offset 1, that is just after the end of field $i$ of $c$, not waiting for the actual end of field $c$. Such an optimization is proposed by [5].
In other words, our formalization generalizes empty base offsets to empty fields, thanks to the fact that the data size of a field of type $C[n]$ is $(n-1) \cdot \mathsf{size}_C + \mathsf{dsize}_C$ instead of $n \cdot \mathsf{size}_C$ as prescribed by Itanium C++ ABI [1].

### 3.3.5 Dynamic type data

If $C$ is dynamic, then there are two cases:

- if $C$ has a direct non-virtual base $A$ that is dynamic, then this base may be chosen as a *primary base* that will have offset 0 within the non-virtual part of $C$.
- otherwise, some space must be explicitly allocated at the beginning of $C$ to store the dynamic type data.

This indeed allows any dynamic class $C$ to necessarily have some space at the beginning of $C$ to store the dynamic type data. If $C$ has a primary base $A$, then $C$ and $A$ will share their dynamic type data.

**Hypothesis 20.** *The size of dynamic type data is denoted* dtdsize*. It is positive and it does not depend on any class.*

In practice, for such compiler as GNU G++, $\mathsf{dtdsize} = \mathbf{sizeof}(\mathbf{void}*)$ to store a pointer to a virtual table. Other compilers like MSVC++ use $\mathsf{dtdsize} = 2 * \mathbf{sizeof}(\mathbf{void}*)$ to store additional data (a pointer to the table of virtual bases) [3].
The choice of the primary base is formalized as follows:

**Hypothesis 21.** *There exists* $\mathsf{pbase}_C \in \mathfrak{P}(\mathcal{DNV}(C))$ *with cardinality 0 or 1 (that is,* $\mathsf{pbase}_C \in \mathsf{option}\ \mathcal{DNV}(C)$*), such that if* $\mathsf{pbase}_C = \{A\}$*, then* $C$ *and* $A$ *are dynamic and:*

$$\mathsf{dnvboff}_C(A) = 0 \tag{C21}$$

*In this case,* $A$ *is called the* primary base *of* $C$*.*

A class $C$ having a dynamic non-virtual direct base does not necessarily have a primary class: the layout algorithm is not forced to choose any. However, as regards performance, such an algorithm could yield under-optimized layouts, with $C$ having its own dynamic type data disjoint from all of its bases'.
We have to prove non-overlapping lemmas for dynamic type data. Indeed, we must ensure that:

- whenever a field is written to, the dynamic type data of dynamic subobjects are not altered
- whenever a subobject is being initialized, the dynamic type data of other subobjects are not altered

### Disjointness of fields and dynamic type data

**Hypothesis 22.** *The dynamic type data of a* dynamic *class* $C$ *is disjoint from the field data of* $C$*:*

$$\mathsf{dtdsize} \leq \mathsf{fboundary}_C \tag{C19}$$

This hypothesis along with (C11, p. 26) allows to show that the dynamic type data of a class is included in its non-virtual data.
The following hypothesis makes $C$'s dynamic type data disjoint from any of its non-empty non-virtual direct bases if $C$ has no primary base:

**Hypothesis 23.** *If* $\mathsf{pbase}_C = \varnothing$*, then for any* non-empty *direct non-virtual base* $B$ *of a dynamic class* $C$*:*

$$\mathsf{dtdsize} \leq \mathsf{dnvboff}_C(B) \tag{C20}$$

**Lemma 25.** *If* $l$ *is a non-virtual path from some dynamic class* $C$ *to some non-empty class* $B$*, then the dynamic type data of* $C$ *is disjoint from the field data of* $B$*. More precisely:*

$$\mathsf{dtdsize} \leq \mathsf{nvsoff}(l) + \mathsf{fboundary}_B$$

*Proof.* By induction on $l$. If $l = C :: \mathsf{nil}$ is the trivial path, then $B = C$ and (C19, p. 34) concludes. Otherwise, if $l = C :: B' :: l'$, then $B'$ is a non-virtual direct base of $C$. There are two cases:

- If $B'$ is the primary base of $C$, having offset 0 within $C$, then $B'$ is dynamic and we use the induction hypothesis.
- Otherwise, it suffices to show that $B'$ itself starts at offset at least $\mathsf{dtdsize}$ within $C$, which would conclude. There are two cases:
  - If $C$ has no primary base, then (C20, p. 34) concludes.
  - Otherwise, if $P$ is the primary base of $C$, then the data of $P$ and $B'$ are disjoint. However, we need an additional hypothesis to conclude:

    **Hypothesis 24.** *Any non-empty class $A$ has non-null non-virtual data:*

    $$0 < \mathit{nvdsize}_A \tag{C22}$$

    In practice, this hypothesis makes sense, but its proof depends on the layout algorithm and, in particular, the notions of empty class and dynamic class.

□

**Lemma 26.** *If $l$ is a non-virtual path from some class $C$ to some dynamic class $B$, then the dynamic type data of $B$ is disjoint from the field data of $C$. More precisely:*

$$\mathit{nvsoff}(l) + \mathit{dtdsize} \leq \mathit{fboundary}_C$$

*Proof.* By case analysis on $l$. If $l = C :: \mathsf{nil}$ is the trivial path, then $B = C$ and (C19, p. 34) concludes. Otherwise, if $l = C :: B' :: l'$, then $B'$ is a non-virtual direct base of $C$, and:

- $\mathsf{dtdsize} \leq \mathsf{fboundary}_B$ by (C19, p. 34)
- $\mathsf{fboundary}_B \leq \mathsf{nvdsize}_B$ by (C11, p. 26)
- $\mathsf{nvsoff}(B' :: l) + \mathsf{nvdsize}_B \leq \mathsf{nvdsize}_{B'}$ by Lemma 17 (p. 27)
- $\mathsf{dnvboff}_C(B') + \mathsf{nvdsize}_{B'} \leq \mathsf{fboundary}_C$ by (C7, p. 26)

which concludes. □

**Lemma 27.** *If $l_1$ is a non-virtual path from $C$ to some dynamic class $B_1$, and if $l_2$ is a non-virtual path from $C$ to some non-empty class $B_2$, then the field data of $B_2$ is disjoint from the dynamic data of $B_1$:*

$$\begin{array}{cc} & [\mathit{nvsoff}(l_1), \mathit{nvsoff}(l_1) + \mathit{dtdsize}) \\ \bot & \left[\mathit{nvsoff}(l_2) + \mathit{fboundary}_{B_2}(F), \mathit{nvsoff}(l_2) + \mathit{nvdsize}_{B_2}\right) \end{array}$$

*Proof.* There are three cases:

- if there is a non-virtual path $p$ from $B_1$ to $B_2$ such that $l_2 = l_1@_{\mathsf{Repeated}}p$, then Lemma 25 (p. 34) concludes.
- if there is a non-virtual path $p$ from $B_2$ to $B_1$ such that $l_1 = l_2@_{\mathsf{Repeated}}p$, then Lemma 26 (p. 35) concludes.
- Otherwise, there is a class $A$ and a non-virtual path $p$ from $C$ to $A$ and $A_1 \neq A_2 \in \mathcal{DNV}_A$ and non-virtual paths $l'_1$ from $A_1$ to $B_1$ and $l'_2$ from $A_2$ to $B_2$ such that $\forall i : \ l_i = p@_{\mathsf{Repeated}}(A :: A_i :: \mathsf{nil})@_{\mathsf{Repeated}}l'_i$. As $A_1$ and $A_2$ are two distinct direct non-virtual non-empty bases of $A$, their data zones are disjoint, which concludes.

□

**Corollary 28.** *If $P_1$ is an inheritance path from $C$ to some dynamic class $B_1$, and if $P_2$ is an inheritance path from $C$ to some non-empty class $B_2$, then the field data of $B_2$ is disjoint from the dynamic data of $B_1$:*

$$\begin{array}{cc} & [\mathit{soff}_C(P_1), \mathit{soff}_C(P_1) + \mathit{dtdsize}) \\ \bot & \left[\mathit{soff}_C(P_2) + \mathit{fboundary}_{B_2}(F), \mathit{soff}_C(P_2) + \mathit{nvdsize}_{B_2}\right) \end{array}$$

*Proof.* Let $P_i = (h_i, B'_i :: l_i)$ for each $i$. Then, there are two cases:

- if $B'_1 = B'_2$, then the previous lemma immediately concludes.
- Otherwise, $B'_1$ and $B'_2$ are two distinct non-empty generalized virtual bases of $C$, so by (C12, p. 30) their data are disjoint, which concludes thanks to Lemma 18 (p. 27)

□

**Theorem 3.** *If $p_1, p_2$ are generalized subobjects of static type $B_1, B_2$ within an array of $C$, such that $B_1$ defines a scalar field $F$ and $B_2$ is dynamic, then $F$ is disjoint from the dynamic type data of $B_1$:*

$$\begin{array}{cc} & [\mathit{off}_C(p_1) + \mathit{foff}_{B_1}(F), \mathit{off}_C(P_1) + \mathit{foff}_{B_1}(F) + \mathit{fdsize}(F)) \\ \bot & [\mathit{off}_C(p_2), \mathit{off}_C(p_2) + \mathit{dtdsize}) \end{array}$$

*Proof.* By induction on $\mathsf{plength}(p_1)$. For each $i$, let $p_i \propto (j_i, P_i, \Phi_i)$. Then:

- If $j_1 \neq j_2$, then we have two different cells of the array of $C$, so thanks to (C4, p. 31) and Lemma 19 (p. 27), their data are disjoint.

Otherwise, let $B'_i$ be the static type of $P_i$. By Lemma 24 (p. 32), $F$ is included in the field data of $B'_1$.

- If $\Phi_2 = \varnothing$, then Corollary 28 (p. 36) concludes.

Otherwise, let $\Phi_2 = \{F_2', p_2'\}$. Then, the dynamic type data of $B_2$ is included in the data zone of the structure field $F_2'$ (by Lemma 19 (p. 27)), which is itself in the field data zone of $B_2'$.

- If $P_1 \neq P_2$, then, by Lemma 21 (p. 30), the two field data zones of $B_1'$ and $B_2'$ are disjoint, which concludes.

Otherwise, $P_1 = P_2$ and $B_1' = B_2'$.

- If $\Phi_1 = \varnothing$, then $F$ and $F_2'$ are distinct fields (because $F$ is scalar whereas $F_2'$ is a structure field) of the same subobject, so by Hypothesis (C9, p. 28) their data are disjoint.

Otherwise, let $\Phi_1 = \{F_1', p_1'\}$ with $\mathsf{plength}(p_1') = \mathsf{plength}(p_1) - 1$ to allow induction.

- If $F_1' \neq F_2'$, then they are distinct fields of the same subobject, so their data are disjoint.
- Otherwise, we may use the induction hypothesis.

□

**Disjointness of two dynamic type data** The non-virtual data of two different non-virtual bases of $C$ are not necessarily disjoint, because of those classes that share their dynamic type data with their primary bases. As this phenomenon can propagate, it is necessary to precisely determine which subobjects will share their dynamic type data.

**Definition 18.** *A non-virtual path $l$ from some class $B$ to some class $A$ is* primary *if, and only if, at least one of the following conditions holds:*

- *either $B = A$ and $l = B :: \mathsf{nil}$*
- *or $B$ has a primary base $B'$ and $l = B :: B' :: l'$ for some $l'$, such that $B' :: l'$ is a primary path from $B'$ to $A$.*

**Lemma 29.** *For any non-virtual, non-primary path $l$ from some class $B$ to some class $A$, there exists a unique path denoted $red(l)$ and called the* reduced path *from $l$, such that there exists $l'$ the longest primary path such that $l = red(l)@_{\mathsf{Repeated}}l'$. By convention, if $l$ is primary, then $red(l) \underset{\text{def.}}{=\!=} B :: \mathsf{nil}$.*

**Lemma 30.** *If $l'$ is a primary path, then $red(l@_{\mathsf{Repeated}}l') = red(l)$.*

**Lemma 31.** *For any non-virtual path $l$, $\mathsf{nvsoff}(l) = \mathsf{nvsoff}(\mathsf{red}(l))$.*
*In particular, any non-virtual path $l$ to a dynamic class shares its dynamic type data with its reduced path $red(l)$.*

*Proof.* It suffices to show that for any non-virtual paths $l$ and $l'$, if $l'$ is primary, then $\mathsf{nvsoff}(l@_{\mathsf{Repeated}}l') = \mathsf{nvsoff}(l)$. By induction on $l'$. □

Conversely:

**Lemma 32.** *If two* non-primary *non-virtual paths $l_1, l_2$ from some class $B$ to some dynamic classes $A_1, A_2$ such that $\mathsf{red}(l_1) \neq \mathsf{red}(l_2)$, then their dynamic type data are disjoint.*

*Proof.* Pose $l'_i = \mathsf{red}(l_i)$ for each $i$. Then, as $l_1$ and $l_2$ are not primary, their reduced paths $l'_1, l'_2$ are not trivial. By symmetry, we may assume $\mathsf{length}(l'_1) \leq \mathsf{length}(l'_2)$. Then, there are two cases:

- If $l'_2 = l'_1@_{\mathsf{Repeated}}l$, then, as $l'_2 \neq l'_1$, $l$ is not trivial. So, $l'_1 = \lambda{+\!\!+}A'_1 :: A_1 :: \mathsf{nil}$ and there are two cases:
  - either $l = A_1 :: A_2 :: \mathsf{nil}$, so that $A_2$ is a non-primary direct non-virtual base of $A_1$, which concludes
  - either $l = A_1 :: \lambda' :: A' :: A_2 :: \mathsf{nil}$ for some non-virtual base $A'$ of $A_1$ such that $A_2$ is a non-primary direct non-virtual base of $A'$. Then, $A_2$ starts at offset at least $\mathsf{dtdsize}$ in $A'$, which is enough to conclude.

□

**Corollary 33.** *If two non-virtual paths $l_1, l_2$ from some class $B$ to some dynamic classes $A_1, A_2$ such that $\mathsf{red}(l_1) \neq \mathsf{red}(l_2)$, then their dynamic type data are disjoint.*

*Proof.*
- If both paths were primary, then they would have the same reduced path, which is absurd.
- If (for instance) $l_1$ is primary and $l_2$ is non-primary, then $A_2$ starts at offset at least $\mathsf{dtdsize}$ in $B$, which concludes.
- Otherwise, Lemma 32 (p. 37) concludes.

□

**Corollary 34.** *If two paths $(h_1, l_1), (h_2, l_2)$ from some class $C$ to some dynamic classes $A_1, A_2$ such that $\mathsf{red}(l_1) \neq \mathsf{red}(l_2)$, then their dynamic type data are disjoint.*

**Theorem 4.** *If $p_1 = (\mathcal{P}_1, j_1, P_1)$, $p_2 = (\mathcal{P}_2, j_2, P_2)$ are two generalized subobjects from $C[n]$ to some dynamic classes $A_1, A_2$ such that $(\mathcal{P}_1, j_1, \mathsf{red}(P_1)) \neq (\mathcal{P}_2, j_2, \mathsf{red}(P_2))$, then their dynamic type data are disjoint.*

*Proof.* As there is a condition on $P_1, P_2$, we cannot use the alternate representation for pointers. So a direct reasoning is necessary.

- If $(\mathcal{P}_1, j_1) = (\mathcal{P}_2, j_2)$, then Corollary 34 (p. 38) concludes.
- Otherwise, if $\mathcal{P}_1 = \mathcal{P}_2$ and $j_1 \neq j_2$, then there are two different cells of the same array, so their data are disjoint.

Otherwise, assume $\mathcal{P}_1 \neq \mathcal{P}_2$. By symmetry, we may assume $\mathsf{length}(\mathcal{P}_1) \leq \mathsf{length}(\mathcal{P}_2)$. Then, we have $\mathcal{P}_2 = (j'_2, P'_2, F'_2) :: \mathcal{P}'_2$. Reason by structural induction on $\mathcal{P}_1$. There are several cases:

- If $\mathcal{P}_1 = \mathsf{nil}$, then the dynamic type data targeted by $p_2$ is included in the data of $F'_2$ which is included in the field data of the subobject $p_1$. Then, Theorem 3 (p. 36) concludes.
- Otherwise, $\mathcal{P}_1 = (j'_1, P'_1, F'_1) :: \mathcal{P}'_1$. If $(j'_1, P'_1) \neq (j'_2, P'_2)$, then our generalized subobjects $p_1$ and $p_2$ are located in disjoint field zones, which concludes. Otherwise, if $F'_1 \neq F'_2$, then the data of those two fields are disjoint, which concludes. Otherwise, we may use the induction hypothesis.

□

**Dynamic type data alignment**

**Hypothesis 25.** *We assume that there exists an alignment* $\mathit{dtdalign} > 0$ *for dynamic type data.*

Then, the following theorem:

**Theorem 5.** *If $p$ is a generalized subobject of static type $A$ from a structure array of $C$, such that $A$ is dynamic, then the access to the dynamic type data of $A$ is correctly aligned:*

$$(\mathit{dtdalign} \mid \mathit{off}_C(p))$$

$$(\mathit{dtdalign} \mid \mathit{align}_C)$$

is a direct consequence of Corollary 16 (p. 24) requiring the following hypothesis:

**Hypothesis 26.** *If $A$ is a dynamic class, then:*

$$(\mathit{dtdalign} \mid \mathit{nvalign}_A) \tag{C16}$$

### 3.3.6 Identity of subobjects

In this section, we want to show that if $p_1 \neq p_2$ are two distinct generalized subobjects of the same static type $A$ from the same structure array of $C$, then they are located at distinct offsets $\mathsf{off}_C(p_1) \neq \mathsf{off}_C(p_2)$ within the structure array.
The proof of this theorem is very different depending on whether $A$ is empty or not.

**Non-empty base offsets** If $A$ is not empty, then (C22, p. 35) ensures that $\mathsf{nvdsize}_A \neq 0$.
However, recall that the non-virtual data of two distinct subobjects of the same static type are not necessarily disjoint, because of their dynamic type data.

**Lemma 35.** *If $l_1, l_2$ are distinct non-virtual paths from some class $B$ to some non-empty class $A$, then $\mathit{nvsoff}(l_1) \neq \mathit{nvsoff}(l_2)$.*

*Proof.* By symmetry, we may assume $\mathsf{length}(l_1) \leq \mathsf{length}(l_2)$. Then, there are two cases:

- Case $l_2 = l_1 @_{\mathsf{Repeated}} l$ is absurd. Indeed, as $l_1$ and $l_2$ have the same static type $A$, we would have $l$ trivial, so $l_2 = l_1$.
- So, $l_1$ and $l_2$ point to subobjects of some classes $B'_1 \neq B'_2$ that are distinct direct non-virtual bases of some class $B'$. As the non-virtual data of $B'_1, B'_2$ are disjoint, and the non-virtual data of $A$ is non-null, arithmetics conclude.

□

**Corollary 36.** *If $P_1, P_2$ are distinct paths from some class $B$ to some non-empty class $A$, then $\mathit{soff}_B(P_1) \neq \mathit{soff}_B(P_2)$.*

**Theorem 6.** *If $p_1, p_2$ are distinct generalized subobjects from some class $C$ to some non-empty class $A$, then $\mathit{off}_C(p_1) \neq \mathit{off}_C(p_2)$.*

*Proof.* For each $i$, let $p_i \propto (j_i, P_i, \Phi_i)$. By symmetry, we may assume $\mathsf{plength}(l_1) \leq \mathsf{plength}(l_2)$. Reason by induction on $\mathsf{plength}(p_1)$. Then, there are several cases:

- if $j_1 \neq j_2$, then we have two distinct cells of the same array, so their data are disjoint.
- Otherwise, if $j_1 = j_2$ and $\Phi_2 = \{(F_2, p'_2)\}$, then there are two cases:
  - if $\Phi_1 = \{(F_1, p'_1)\}$, then there are three cases:
    - if $P_1 = P_2$, then either $F_1 = F_2$, in which case we may use the induction hypothesis, or $F_1 \neq F_2$ so those are two distinct fields of the same subobject, so their data are disjoint.
    - otherwise, the designated subobjects are located in disjoint field zones.
  - Otherwise, $\Phi_1 = \varnothing$. Then, there are several cases:
    - Case $P_2 = P_1 @_{\mathsf{Repeated}} P$ is absurd (the hierarchy is well-founded).

* Case $P_1 = P_2@_{\mathsf{Repeated}}(B_2 :: B' :: Q)$: in that case, whereas $p_2$ is located in the field data zone of $B_2$, $p_1$ is located in the data zone of $B'$ which is a direct non-virtual base of $B_2$: those two data zones are disjoint
* Otherwise, $P_1 = L@_{\mathsf{Repeated}}B' :: B'_1 :: Q_1$ and $P_2 = L@_{\mathsf{Repeated}}B' :: B'_2 :: Q_2$ with $B'_1$ and $B'_2$ two distinct direct non-virtual bases of $B'$, so their data are disjoint.

- Otherwise, $j_1 = j_2$, $\Phi_2 = \varnothing$ and $\Phi_1 = \varnothing$ (as $\mathsf{plength}(p_1) \leq \mathsf{plength}(p_2)$), so Corollary 36 (p. 40) concludes.

□

**Empty base offsets** Unfortunately, this kind of reasoning cannot be done if $A$ is empty. In this case, we have to explicitly enumerate the offsets to all possible subobjects of an empty static type.

**Definition 19.** *The sets* $\mathit{nveboffs}_C$ *of the* non-virtual empty base offsets *of* $C$ *and* $\mathit{eboffs}_C$ *of* empty base offsets *of* $C$ *are defined as follows:*

$$\begin{aligned}
\mathit{nveboffs}_C \underset{\text{def.}}{=\!=} & \begin{cases} \{(C,0)\} & \text{if } C \text{ is empty} \\ \varnothing & \text{otherwise} \end{cases} \\
& \cup \bigcup_{B \in \mathcal{NV}(C)} \mathit{dnvboff}_C(B) + \mathit{nveboffs}_B \\
& \cup \bigcup_{(f,B,n) \in St\mathcal{F}(C)} \bigcup_{i \in [0 \ldots n)} \mathit{foff}_C(f,B,n) + i \cdot \mathit{size}_B + \mathit{eboffs}_B
\end{aligned}$$

$$\mathit{eboffs}_C \underset{\text{def.}}{=\!=} \bigcup_{B \in \tilde{mathcalV}(C)} \mathit{nveboffs}_B$$

In our Coq development, those two sets are defined as mutually inductive predicates. Only at the level of the algorithms, we show that we can construct for any class $C$ two sets such that $(A, o)$ is in either set if and only if $(A, o)$ verifies the corresponding predicate, assuming that such sets exist for any class $B \prec C$. Indeed, we shall see that those sets are computed at the same time as class layout.

**Lemma 37.** $(A, o) \in \mathit{nveboffs}_C$ *if, and only if,* $A$ *is empty and there is a class* $B$ *and a non-virtual path* $p$ *from* $C$ *to* $B$ *such that at least one of the following conditions holds:*

- *either* $B = A$ *and* $o = \mathit{nvsoff}(p)$

- *or there is a structure array field $f$ of some type $B'[m']$ defined in class $B$ and an array path $\mathcal{P}$ from $B'[m']$ to some $A'[n']$, a nonnegative integer $k < n'$ and an inheritance path $Q$ from $A'$ to $A$ such that:*

$$o = \mathit{nvsoff}(p) + \mathit{aoff}_{B'}(\mathcal{P}) \ + k \cdot \mathit{size}_{A'} \ + \mathit{soff}_{A'}(Q)$$

*$(A, o) \in \mathsf{eboffs}_C$ if, and only if, $A$ is empty and there is a class $B$ and an inheritance path $P$ from $C$ to $B$ such that at least one of the following conditions holds:*

- *either $B = A$ and $o = \mathit{soff}_C(P)$*
- *or there is a structure array field $f$ of some type $B'[m']$ defined in class $B$ and an array path $\mathcal{P}$ from $B'[m']$ to some $A'[n']$, a nonnegative integer $k < n'$ and an inheritance path $Q$ from $A'$ to $A$ such that:*

$$o = \mathit{soff}_C(P) + \mathit{aoff}_{B'}(\mathcal{P}) \ + k \cdot \mathit{size}_{A'} \ + \mathit{soff}_{A'}(Q)$$

*Proof.* $\Rightarrow$: by mutual induction on the definitions of $\mathsf{eboffs}$ and $\mathsf{nveboffs}$.
$\Leftarrow$: cases $B = A$ by induction on $p$ for non-virtual inheritance, by case analysis on $h$ where $P = (h, p)$ for virtual inheritance. Other cases by induction on $\mathcal{P}$. □

**Theorem 7.** *Let $A$ be an empty class. Two different pointers of static type $A$, but from the same initial object, point to a different memory location, if and only if, in the construction of the sets $\mathsf{eboffs}$ and $\mathsf{nveboffs}$, the unions are disjoint, i.e. for all classes $C$, the following conditions hold:*

- $\forall B_1, B_2 \in \mathcal{DNV}(C) : B_1 \neq B_2 \Rightarrow$

$$\mathit{dnvboff}_C(B_1) + \mathsf{nveboffs}_{B_1} \perp \mathit{dnvboff}_C(B_2) + \mathsf{nveboffs}_{B_2} \qquad \text{(C23)}$$

- $\forall B_1 \in \mathcal{DNV}(C), \forall (f, B_2, n) \in \mathit{St}\mathcal{F}(C), \forall i \in [0 \ldots (n-1)] :$

$$\mathit{dnvboff}_C(B_1) + \mathsf{nveboffs}_{B_1} \perp \mathit{foff}_C(f, B_2, n) + i \cdot \mathit{size}_{B_2} + \mathsf{eboffs}_{B_2} \qquad \text{(C24)}$$

- $\forall (f_1, B_1, n_1), (f_2, B_2, n_2) \in \mathit{St}\mathcal{F}(C), (f_1, B_1, n_1) \neq (f_2, B_2, n_2) \Rightarrow \forall i_1 \in [0 \ldots (n_1 - 1)], \forall i_2 \in [0 \ldots (n_2 - 1)] :$

$$\mathit{foff}_C(f_1, B_1, n_1) + i_1 \cdot \mathit{size}_{B_1} + \mathsf{eboffs}_{B_1} \perp \mathit{foff}_C(f_2, B_2, n_2) + i_2 \cdot \mathit{size}_{B_2} + \mathsf{eboffs}_{B_2} \qquad \text{(C25)}$$

- $\forall B_1, B_2 \in \tilde{\mathcal{V}}(C) : B_1 \neq B_2 \Rightarrow$

$$\mathit{vboff}_C(B_1) + \mathsf{nveboffs}_{B_1} \perp \mathit{vboff}_C(B_2) + \mathsf{nveboffs}_{B_2} \qquad \text{(C26)}$$

*Proof.* Mostly by definition of nveboffs and eboffs, except for two empty bases of distinct cells of a structure array. In that case, we have to use the fact that two different cells of the same array are totally disjoint (not only their data). Then, we need a further hypothesis:

**Hypothesis 27.** *The non-virtual size of any class $B$ is positive.*

$$\mathsf{nvsize}_C > 0 \tag{C5}$$

□

The reason why (C5, p. 43) is required already for non-virtual sizes (not only for total sizes) may seem not obvious, but the following simple example makes it clear:

```
struct A           {};
struct B0: A       {};
struct B1: A       {};
struct C:  B0, B1  {};

C c[2];
A * a0 = (A*) (B1*) &c[0];
A * a1 = (A*) (B0*) &c[1];
```

If $\mathsf{nvsize}_A = 0$, $\mathsf{nvsize}_{B_1} = 0$, and $\mathsf{nvsize}_{B_2} = 0$ were allowed, then we would not be able to distinguish pointers $a_0$ and $a_1$ above.

In practice, following the behaviour prescribed by Itanium C++ ABI [1] to compute offsets for a given class $C$, assuming that layout has already been computed for all bases of $C$ and all classes that are types of some structure fields of $C$, components of $C$ (base or field) are laid out one after another, and for each attempt to lay out a given component, it is checked against all components previously laid out.
Considering intervals instead of eboffs and nveboffs is not enough, even in practice. Indeed, consider the following hierarchy:

```
struct A              {};
struct B              {};
struct C:  A          {};
struct D:  B          {};
struct E1: A,  B      {};
struct E2: A,  B      {};
struct F:  E1, C, E2  {};
struct G:  F,  D      {};
```

Then, the layout of G given by GNU G++ is such that:

$$\begin{array}{lll}
\mathsf{dnvboff}_C(A) = 0 & & \\
\mathsf{dnvboff}_D(B) = 0 & & \\
\mathsf{dnvboff}_{E_1}(A) = 0 & \mathsf{dnvboff}_{E_1}(B) = 0 & \\
\mathsf{dnvboff}_{E_2}(A) = 0 & \mathsf{dnvboff}_{E_2}(B) = 0 & \\
\mathsf{dnvboff}_F(E_1) = 0 & \mathsf{dnvboff}_F(C) = 1 & \mathsf{dnvboff}_F(E_2) = 2 \\
\mathsf{dnvboff}_G(F) = 0 & &
\end{array}$$

If the type conflict test (whose existence, but not the way it should work, is prescribed by Itanium C++ ABI [1]) relies on interval checking, then base $F$ of $G$ would define interval $[0 \ldots 2]$ for base $B$, so that base $D$ of $G$ would be laid out at offset 3. However, it would be consistent to lay out $D$ at offset 1 within $G$, which GNU G++ actually does. This justifies why the exact sets of offsets have to be checked.

# 4 Implementation of realistic layout algorithms (`CPP.v`, `CommonVendorABI.v`, `CCCPP.v`)

We chose not to directly impose a layout, as Compcert does with C structures, but to set some constraints for layout algorithms, allowing them to make some optimizations that our constraints do not necessarily catch, such as field or base reordering for optimizing data alignment.
We implemented two algorithms: one based on the Itanium C++ ABI [1], and another algorithm optimized for empty base tail padding.
For both algorithms, we took a more precise definition of dynamic classes (`CPP.v`):

**Definition 20.** *A class $C$ is* dynamic *if, and only if, at least one of the following conditions holds:*

- *$C$ has a virtual method*
- *$C$ has a virtual base*
- *$C$ has a dynamic non-virtual direct base*

Under this definition, it is consistent to reserve no additional space for non-dynamic classes, thanks to the following limitations of the C++ Standard:

- static base-to-derived cast cannot go through virtual inheritance
- dynamic cast cannot be used from a class without virtual methods (a fortiori from a non-dynamic class)

## 4.1 An algorithm based on the Itanium C++ ABI

The principle of this algorithm based on the common vendor C++ ABI for Itanium is that all bases and fields are laid out within a class such that they are mutually *totally* disjoint, not only their data. This actually incurs stronger constraints than the ones stated in the previous section. So, for two such disjoint components (bases or fields), any two offsets to generalized subobjects of some class type $A$ (empty or not) from those components are automatically distinct.
But such a layout would be too naive for empty bases, as it would consume some space for them. For this reason, an important exception is prescribed for empty bases, which may be laid out at offset 0. In that case, explicit checks are necessary to ensure that a subobject reachable from an empty base will not conflict with another subobject of the same type reachable from another base or field.
However, to algorithmically limit the need for such explicit checks, this algorithm considers classes with fields to be non-empty. More formally:

**Definition 21.** *A class $C$ is* empty *if, and only if, all the following conditions hold:*

- *$C$ has no virtual methods*
- *$C$ has no direct virtual base*
- *$C$ has no fields*
- *all direct non-virtual bases of $C$ are empty*

In particular, a class having a virtual base is not empty. More generally:

**Lemma 38.** *Dynamic classes are not empty.*

Then, each class $C$ is laid out through the following way, assuming that all classes $B$ such that $B \prec C$ have been already laid out:

1. If $C$ has a dynamic direct non-virtual base $B$, then choose $B$ as the primary base of $C$, and lay out $B$ within $C$ at offset 0.
2. Otherwise, if $C$ is dynamic, then reserve some space for dynamic type data
3. For each direct non-virtual base $B$ of $C$ that is not the primary base of $B$, lay it out within $C$ as follows:
    - If $B$ is empty, try to lay it out at offset 0, unless there is a type conflict for empty bases.

- Otherwise, try to lay out $B$ at the current value of $\mathsf{nvdsize}_C$ so far. If there is a type conflict for empty bases, then increase by the non-virtual alignment $\mathsf{nvalign}_B$ of the base $B$, knowing that beyond the current value for $\mathsf{nvsize}_C$, there will be no conflict (because all offsets of bases reachable from $C$ so far are less than $\mathsf{nvsize}_C$).
- If $B$ is not empty, update $\mathsf{nvdsize}_C$ to include the *whole* non-virtual size of the base (not only its non-virtual data size).

4. Then, for each field, lay it out at increasing offsets, making them *wholly* disjoint (not only their data).

5. Then, lay out all virtual bases of $C$ the same way as for non-virtual bases (i.e. trying offset 0 for empty virtual bases, and making them wholly disjoint – not only their data).

During the layout of the components of $C$, it is also wise to construct the sets $\mathsf{nveboffs}_C$ and $\mathsf{eboffs}_C$ of generalized subobjects of an empty class type that are reachable from $C$.
However, the overall shape of the algorithm as described above and in the Itanium C++ ABI does not exactly state how type conflicts for empty bases should be resolved. In fact, there are only two possible conflicts:

- when trying to lay out an empty base $B$ at offset 0, it is necessary to check that no empty base reachable through $B$ conflict with any empty base reachable from all other non-virtual bases that have been already laid out.

- when trying to lay out a non-empty base $B$, or a field $f$, it is only necessary to check that there is no conflict with any empty base reachable from *non-virtual empty bases* laid out so far. Indeed, this algorithm lays out all non-empty components in such way that two distinct non-empty components are *totally* disjoint (not only their data).

More formally:

**The main algorithm** The general shape of the algorithm is as follows. Starting with the choice of the primary base and/or the allocation of dynamic type data, it then lays out the remaining components of the class, first the non-virtual bases (other than the primary base), then the fields, and finally the virtual bases.

$\mathsf{nvdsize}_C \leftarrow 0,$
$\mathsf{pbase}_C, \mathsf{nveboffs}_C, \mathsf{eboffs}'_C \leftarrow \varnothing,$
$\mathsf{dnvboff}_C, \mathsf{foff}_C, \mathsf{vboff}_C : \varnothing \to \mathbb{Z},$
$\mathsf{nvsize}_C, \mathsf{nvalign}_C \leftarrow 1$

**if** $C$ is dynamic **then**
  **if** $\exists B$ dynamic direct non-virtual base of $C$ **then**
    $\mathsf{pbase}_C \leftarrow \{B\}$
    $\mathsf{dnvboff}_C(B) \leftarrow 0$
    $\mathsf{nveboffs}_C \leftarrow \mathsf{nveboffs}_B$
    $\mathsf{nvdsize}_C \leftarrow \mathsf{nvsize}_B$
    $\mathsf{nvsize}_C \leftarrow \mathsf{nvsize}_B$
    $\mathsf{nvalign}_C \leftarrow \mathsf{nvalign}_B$
  **else** {there is no dynamic direct non-virtual base of $C$}
    $\mathsf{nvdsize}_C \leftarrow \mathsf{dtdsize}$
    $\mathsf{nvsize}_C \leftarrow \mathsf{dtdsize}$
    $\mathsf{nvalign}_C \leftarrow \mathsf{dtdalign}$
  **end if**
**end if**
lay out direct non-virtual bases $B$ of $C$ such that $\mathsf{pbase}_C \neq \{B\}$
$\mathsf{fboundary}_C \leftarrow \mathsf{nvdsize}_C$
lay out fields of $C$
$\mathsf{eboffs}_C \leftarrow \mathsf{nveboffs}_C$
$\mathsf{dsize}_C \leftarrow \mathsf{nvdsize}_C$
$\mathsf{size}_C \leftarrow \mathsf{nvsize}_C$
$\mathsf{align}_C \leftarrow \mathsf{nvalign}_C$
lay out virtual bases of $C$
$\mathsf{vboff}_C(C) \leftarrow 0$
$\mathsf{size}_C \leftarrow \min\{o \mid (\mathsf{align}_C \mid o) \wedge \mathsf{size}_C \leq o\}$

**Direct non-virtual bases**

**for all** $B$ direct non-virtual base of $C$ such that $\mathsf{pbase}_C \neq \{B\}$ **do**
  **if** $B$ is empty and $\mathsf{nveboffs}_B \perp \mathsf{nveboffs}_C$ **then**
    $\mathsf{dnvboff}_C(B) \leftarrow 0$
  **else**
    $\mathsf{dnvboff}_C(B) \leftarrow \min\{o \mid (\mathsf{nvalign}_B \mid o) \wedge \mathsf{nvdsize}_C \leq o\}$
    **while** $\mathsf{dnvboff}_C(B) < \mathsf{nvsize}_C \wedge \neg \mathsf{dnvboff}_C(B) + \mathsf{nveboffs}_B \perp \underline{\mathsf{eboffs}'_C}$
    **do**
      $\mathsf{dnvboff}_C(B) \leftarrow \mathsf{dnvboff}_C(B) + \mathsf{nvalign}_B$
    **end while**
  **end if**
  $\mathsf{nveboffs}_C \leftarrow \mathsf{nveboffs}_C \cup \mathsf{dnvboff}_C(B) + \mathsf{nveboffs}_B$
  **if** $B$ is empty **then**
    $\mathsf{eboffs}'_C \leftarrow \mathsf{eboffs}'_C \cup \mathsf{dnvboff}_C(B) + \mathsf{nveboffs}_B$
  **else**
    $\mathsf{nvdsize}_C \leftarrow \mathsf{dnvboff}_C(B) + \underline{\mathsf{nvsize}_B}$
  **end if**
  $\mathsf{nvsize}_C \leftarrow \max(\mathsf{nvsize}_C, \mathsf{dnvboff}_C(B) + \mathsf{nvsize}_B)$

$\mathsf{nvalign}_C \leftarrow \mathrm{lcm}(\mathsf{nvalign}_C, \mathsf{nvalign}_B)$
**end for**

Line 14 imposes every non-empty direct non-virtual base $B$ to have its whole non-virtual part (not only its data) included in the non-virtual *data* of $C$. Thanks to this requirement, conflict checks for empty bases are limited to those offsets reachable only through empty direct bases (line 6). Indeed, in the "else" case of lines 5–8, $B$ is laid out wholly (not only by its data) disjointly from all other non-empty non-virtual bases laid out so far. So, in particular, offsets to empty bases are disjoint.

**Fields**

**for all** $f$ field of $C$ **do**
$\mathsf{foff}_C(f) \leftarrow \min\{o \mid (\mathsf{falign}(f) \mid o) \wedge \mathsf{nvdsize}_C \leq o\}$
**if** $f$ is a structure field $(fid, B, n)$ **then**
**while** $\neg \bigcup_{0 \leq j < n} \mathsf{foff}_C(f) + j \cdot \mathsf{size}_B + \mathsf{eboffs}_B \perp \mathsf{eboffs}'_C$ **do**
$\mathsf{foff}_C(f) \leftarrow \mathsf{foff}_C(f) + \mathsf{align}_B$
**end while**
$\mathsf{nveboff}_C \leftarrow \mathsf{nveboff}_C \cup \bigcup_{0 \leq j < n} \mathsf{foff}_C(f) + j \cdot \mathsf{size}_B + \mathsf{eboffs}_B$
**end if**
$\mathsf{nvdsize}_C \leftarrow \mathsf{foff}_C(f) + \mathsf{fsize}(f)$
$\mathsf{nvsize}_C \leftarrow \max(\mathsf{nvsize}_C, \mathsf{foff}_C(f) + \mathsf{fsize}(f))$
$\mathsf{nvalign}_C \leftarrow \mathrm{lcm}(\mathsf{nvalign}_C, \mathsf{falign}(f))$
**end for**

**Virtual bases**

**for all** $B$ virtual base of $C$ **do**
**if** $B$ is empty and $\mathsf{nveboffs}_B \perp \mathsf{eboffs}_C$ **then**
$\mathsf{vboff}_C(B) \leftarrow 0$
**else**
$\mathsf{vboff}_C(B) \leftarrow \min\{o \mid (\mathsf{align}_B \mid o) \wedge \mathsf{dsize}_C \leq o\}$
**while** $\mathsf{vboff}_C(B) < \mathsf{size}_C \wedge \neg \mathsf{vboff}_C(B) + \mathsf{nveboffs}_B \perp \mathsf{eboffs}'_C$ **do**
$\mathsf{vboff}_C(B) \leftarrow \mathsf{vboff}_C(B) + \mathsf{nvalign}_B$
**end while**
**end if**
$\mathsf{eboffs}_C \leftarrow \mathsf{eboffs}_C \cup \mathsf{vboff}_C(B) + \mathsf{nveboffs}_B$
**if** $B$ is empty **then**
$\mathsf{eboffs}'_C \leftarrow \mathsf{eboffs}'_C \cup \mathsf{vboff}_C(B) + \mathsf{nveboffs}_B$
**else**
$\mathsf{dsize}_C \leftarrow \mathsf{vboff}_C(B) + \underline{\mathsf{nvsize}_B}$
**end if**
$\mathsf{size}_C \leftarrow \max(\mathsf{size}_C, \mathsf{vboff}_C(B) + \mathsf{size}_B)$

$\mathsf{align}_C \leftarrow \mathrm{lcm}(\mathsf{align}_C, \mathsf{nvalign}_B)$
**end for**

**Lemma 39** (Strong invariants)**.** *This algorithm ensures that for any class $C$:*

$$\mathit{nvdsize}_C \leq \mathit{nvsize}_C$$

$$\begin{array}{cc} & [\mathit{foff}_C(F_1), \mathit{foff}_C(F_1) + \underline{\mathit{fsize}}(F_1)) \\ \bot & [\mathit{foff}_C(F_2), \mathit{foff}_C(F_2) + \mathit{fsize}(F_2)) \end{array}$$

Indeed, for this algorithm, non-empty bases and fields are laid out by ensuring that their whole sizes (not only their data) are disjoint.

**Theorem 8.** *This algorithm satisfies the soundness conditions C1 to C26.*

## 4.2 An optimized algorithm

**Definition 22.** *A class $C$ is* empty *if, and only if, all the following conditions hold:*

- *$C$ has no virtual methods*
- *$C$ has no direct virtual base*
- *$C$ has no scalar fields*
- *if $C$ has a structure array field of some type $B$, then $B$ is empty*
- *all direct non-virtual bases of $C$ are empty*

Given the definition of dynamic classes taken for our algorithms, this definition for empty bases is the smallest possible satisfying the constraints given at Hypothesis 2.

**Lemma 40.** *Dynamic classes are not empty.*

In particular, a class having an empty virtual base is not empty.

**The main algorithm** The general shape of the algorithm is similar to the previous one, except when choosing a primary base: in that case, after the choice of the primary base but before the layout of the other direct non-virtual bases, the non-virtual data size is updated to the non-virtual *data* size of the primary base, not its whole non-virtual size.

$\mathsf{nvdsize}_C \leftarrow 0$,
$\mathsf{pbase}_C, \mathsf{nveboffs}_C \leftarrow \varnothing$,
$\mathsf{dnvboff}_C, \mathsf{foff}_C : \varnothing \to \mathbb{Z}$,
$\mathsf{vboff}_C : C \mapsto 0$,
$\mathsf{nvsize}_C, \mathsf{nvalign}_C \leftarrow 1$
**if** $C$ is dynamic **then**

**if** $\exists B$ dynamic direct non-virtual base of $C$ **then**
$\mathsf{pbase}_C \leftarrow \{B\}$
$\mathsf{dnvboff}_C(B) \leftarrow 0$
$\mathsf{nveboffs}_C \leftarrow \mathsf{nveboffs}_B$
$\underline{\mathsf{nvdsize}_C \leftarrow \mathsf{nvdsize}_B}$
$\mathsf{nvsize}_C \leftarrow \mathsf{nvsize}_B$
$\mathsf{nvalign}_C \leftarrow \mathsf{nvalign}_B$
**else** {there is no dynamic direct non-virtual base of $C$}
$\mathsf{nvdsize}_C \leftarrow \mathsf{dtdsize}$
$\mathsf{nvsize}_C \leftarrow \mathsf{dtdsize}$
$\mathsf{nvalign}_C \leftarrow \mathsf{dtdalign}$
**end if**
**end if**
lay out direct non-virtual bases $B$ of $C$ such that $\mathsf{pbase}_C \neq \{B\}$
$\mathsf{fboundary}_C \leftarrow \mathsf{nvdsize}_C$
lay out fields of $C$
$\mathsf{eboffs}_C \leftarrow \mathsf{nveboffs}_C$
$\mathsf{dsize}_C \leftarrow \mathsf{nvdsize}_C$
$\mathsf{size}_C \leftarrow \mathsf{nvsize}_C$
$\mathsf{align}_C \leftarrow \mathsf{nvalign}_C$
lay out virtual bases of $C$
$\underline{\mathsf{size}_C \leftarrow \min\{o \mid (\mathsf{align}_C \mid o) \wedge \mathsf{size}_C \leq o \wedge \mathsf{dsize}_C \leq o\}}$

**Direct non-virtual bases** The algorithm is
**for all** $B$ direct non-virtual base of $C$ such that $\mathsf{pbase}_C \neq \{B\}$ **do**
**if** $\underline{B \text{ is empty}}$ **then**
$\underline{\mathsf{dnvboff}_C(B) \leftarrow 0}$
**else**
$\mathsf{dnvboff}_C(B) \leftarrow \min\{o \mid (\mathsf{nvalign}_B \mid o) \wedge \mathsf{nvdsize}_C \leq o\}$
**end if**
**while** $\underline{\mathsf{dnvboff}_C(B) < \mathsf{nvsize}_C \wedge \neg\mathsf{dnvboff}_C(B) + \mathsf{nveboffs}_B \perp \mathsf{eboffs}_C}$ **do**

$\mathsf{dnvboff}_C(B) \leftarrow \mathsf{dnvboff}_C(B) + \mathsf{nvalign}_B$
**end while**
$\mathsf{nveboffs}_C \leftarrow \mathsf{nveboffs}_C \cup \mathsf{dnvboff}_C(B) + \mathsf{nveboffs}_B$
**if** $\underline{B \text{ is not empty}}$ **then**
$\underline{\mathsf{nvdsize}_C \leftarrow \mathsf{dnvboff}_C(B) + \mathsf{nvdsize}_B}$
**end if**
$\mathsf{nvsize}_C \leftarrow \max(\mathsf{nvsize}_C, \mathsf{dnvboff}_C(B) + \mathsf{nvsize}_B)$
$\mathsf{nvalign}_C \leftarrow \mathrm{lcm}(\mathsf{nvalign}_C, \mathsf{nvalign}_B)$
**end for**

**Fields**

**for all** $f$ field of $C$ **do**
  $\mathsf{foff}_C(f) \leftarrow \min\{o \mid (\mathsf{falign}(f) \mid o) \wedge \mathsf{nvdsize}_C \leq o\}$
  **if** $f$ is a structure field $(fid, B, n)$ **then**
    **if** $B$ is empty **then**
      $\mathsf{foff}_C(f) \leftarrow 0$
    **else**
      $\mathsf{foff}_C(f) \leftarrow \min\{o \mid (\mathsf{align}_B \mid o) \wedge \mathsf{nvdsize}_C \leq o\}$
    **end if**
    **while** $\neg \bigcup_{0 \leq j < n} \mathsf{foff}_C(f) + j \cdot \mathsf{size}_B + \mathsf{eboffs}_B \perp \mathsf{eboffs}_C$ **do**
      $\mathsf{foff}_C(f) \leftarrow \mathsf{foff}_C(f) + \mathsf{align}_B$
    **end while**
    $\mathsf{nveboff}_C \leftarrow \mathsf{nveboff}_C \cup \bigcup_{0 \leq j < n} \mathsf{foff}_C(f) + j \cdot \mathsf{size}_B + \mathsf{eboffs}_B$
    **if** $B$ is not empty **then**
      $\mathsf{nvdsize}_C \leftarrow \mathsf{foff}_C(f) + \mathsf{fdsize}(f)$
    **end if**
  **else** {$f$ is a scalar field}
    $\mathsf{nvdsize}_C \leftarrow \mathsf{foff}_C(f) + \mathsf{fdsize}(f)$
  **end if**
  $\mathsf{nvsize}_C \leftarrow \max(\mathsf{nvsize}_C, \mathsf{foff}_C(f) + \mathsf{fsize}(f))$
  $\mathsf{nvalign}_C \leftarrow \mathrm{lcm}(\mathsf{nvalign}_C, \mathsf{falign}(f))$
**end for**

**Virtual bases**

**for all** $B$ virtual base of $C$ **do**
  **if** $B$ is empty **then**
    $\mathsf{vboff}_C(B) \leftarrow 0$
  **else**
    $\mathsf{vboff}_C(B) \leftarrow \min\{o \mid (\mathsf{align}_B \mid o) \wedge \mathsf{dsize}_C \leq o\}$
  **end if**
  **while** $\mathsf{vboff}_C(B) < \mathsf{size}_C \wedge \neg \mathsf{vboff}_C(B) + \mathsf{nveboffs}_B \perp \mathsf{eboffs}_C$ **do**
    $\mathsf{vboff}_C(B) \leftarrow \mathsf{vboff}_C(B) + \mathsf{nvalign}_B$
  **end while**
  $\mathsf{eboffs}_C \leftarrow \mathsf{eboffs}_C \cup \mathsf{vboff}_C(B) + \mathsf{nveboffs}_B$
  **if** $B$ is not empty **then**
    $\mathsf{dsize}_C \leftarrow \mathsf{vboff}_C(B) + \mathsf{nvdsize}_B$
  **end if**
  $\mathsf{size}_C \leftarrow \max(\mathsf{size}_C, \mathsf{vboff}_C(B) + \mathsf{size}_B)$
  $\mathsf{align}_C \leftarrow \mathrm{lcm}(\mathsf{align}_C, \mathsf{nvalign}_B)$
**end for**

## References


[1] Itanium C++ ABI. `http://www.codesourcery.com/public/cxx-abi/`, March 2001.

[2] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[3] Juan Chen. A typed intermediate language for compiling multiple inheritance. *SIGPLAN Not.*, 42(1):25–30, 2007.

[4] The C++ Standard Committee. ISO/IEC 14882:2003 – The C++ Programming Language.

[5] Nathan C. Myers. The "Empty Member" C++ Optimization. *C++ Issue, Dr. Dobb's Journal*, 1997.

[6] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. Formal verification of object layout for c++ multiple inheritance. Submitted to POPL 2011.

[7] Jonathan G. Rossie, Jr., and Daniel P. Friedman. An algebraic semantics of subobjects. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, pages 187–199. ACM Press, 1996.

[8] Bjarne Stroustrup. *A history of C++: 1979–1991*, pages 699–769. ACM, New York, NY, USA, 1996.

[9] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 345–362, New York, NY, USA, 2006. ACM.