Formal Verification of Object Layout for C++ Multiple Inheritance

Tahina Ramananandro[1] Gabriel Dos Reis[2] Xavier Leroy[1]

[1]INRIA Paris-Rocquencourt [2]Texas A&M University

January 26th, 2011

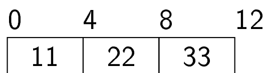
Representing homogeneous data in memory

$$\begin{bmatrix} 11 & 22 & 33 \end{bmatrix}$$

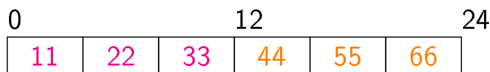
0	4	8	12
11	22	33	

Representing homogeneous data in memory

$$\begin{bmatrix} 11 & 22 & 33 \end{bmatrix}$$

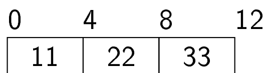


$$\begin{bmatrix} 11 & 22 & 33 \\ 44 & 55 & 66 \end{bmatrix}$$

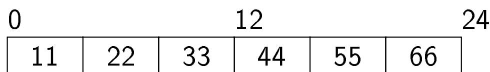


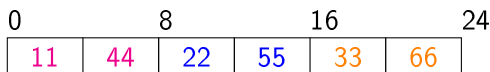
Representing homogeneous data in memory

$$\begin{bmatrix} 11 & 22 & 33 \end{bmatrix}$$



$$\begin{bmatrix} 11 & 22 & 33 \\ 44 & 55 & 66 \end{bmatrix}$$

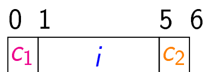




Representing heterogeneous data in memory

struct S { char c_1; int i; char c_2; };

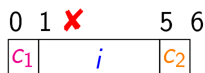
- A naive representation:



Representing heterogeneous data in memory

struct S { char c_1; int i; char c_2; };

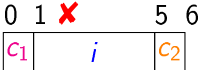
- A naive representation:



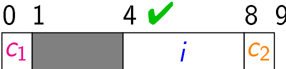
Representing heterogeneous data in memory

struct S { char c_1; int i; char c_2; };

- A naive representation:



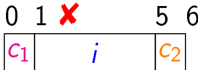
- Correct field alignment requires padding:



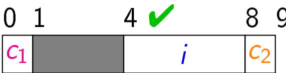
Representing heterogeneous data in memory

struct S { char c_1; int i; char c_2; };

- A naive representation:



- Correct field alignment requires padding:

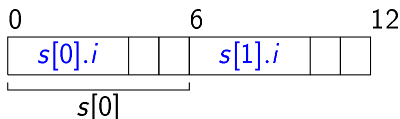


- Reorder fields to save space:

Representing heterogeneous data in memory

struct S { char c_1; int i; char c_2; };

- Making arrays of structures:



Representing heterogeneous data in memory

struct S { char c_1; int i; char c_2; };

- Making arrays of structures:

0 | 6 ✘ | 12

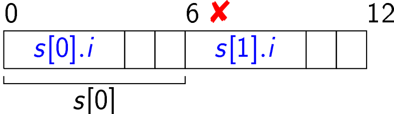
$s[0].i$			$s[1].i$		

$s[0]$

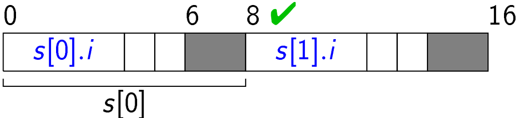
Representing heterogeneous data in memory

struct S { char c_1; int i; char c_2; };

- Making arrays of structures:



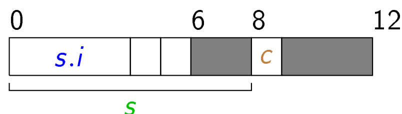
- Correct array cell alignment requires tail padding:



Representing heterogeneous data in memory

```
struct S { char c1;       int i;   char c2; };
struct T { struct S s;    char c;           };
```

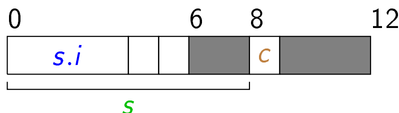
- A naive attempt:



Representing heterogeneous data in memory

struct S { char c_1; int i; char c_2; };
struct T { struct S s; char c; };

- A naive attempt:



- Reuse tail padding in s to store c:

Representing heterogeneous data in memory

```
struct  S  {  char c1;        int i;    char c2;   };
struct  T  {  struct S s;  char c;                };
```

- A naive attempt:

0 6 8 12

s.i			■	c	■

s

- Reuse tail padding in `s` to store `c`:

0 6 8

s.i			c	■

- Reorder fields in `s` to reuse inside padding:

0 2 4 8

		c	■	s.i

$s.c_1$

Focus of our work

A study of data layout for C++ objects.

C++ multiple inheritance

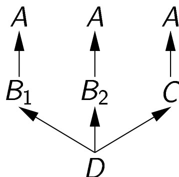
C++ supports object-oriented features:

- Objects can contain scalar fields
- Objects can contain embedded structure fields

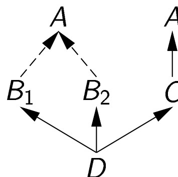
C++ multiple inheritance

C++ supports object-oriented features:

- Objects can contain scalar fields
- Objects can contain embedded structure fields
- Objects can contain *inheritance subobjects*.



repeated (*non-virtual*) inheritance



shared (*virtual*) inheritance

C++ multiple inheritance issues on data layout

Usual layout problems:

- alignment padding
- embedded structures: possibility of reusing padding?

C++ multiple inheritance issues on data layout

Usual layout problems:

- alignment padding
- embedded structures: possibility of reusing padding?

Issues raised by multiple inheritance:

- Dynamic type data (e.g. pointers to virtual tables)
  - needed for dynamic cast, virtual function dispatch
  - even field accesses through virtual inheritance
  - not ordinary fields, may be shared between subobjects
- Object identity: two pointers to different subobjects of the same type must compare different, even in the presence of empty bases.

Empty classes

- No data: no fields, no dynamic type data
- Widely used with STL and Boost libraries
- In practice, compile-time information to tag structures
- Usually minimal footprint with clever compilers

```
struct MyGreater:
              std::binary_function<int, int, bool> {
  bool operator()(int i, int j) const {
    return i > j;
  }
};
```

Empty base layout

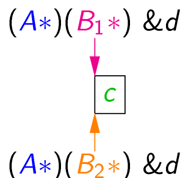
- Minimal size storage, possibly zero (*empty base optimization*)
- Object identity may require further padding

```
struct A { };
struct B1 : A { };
struct B2 : A { };
struct D : B1, B2 { char c; };
D d ;
```

Empty base layout

- Minimal size storage, possibly zero (*empty base optimization*)
- Object identity may require further padding

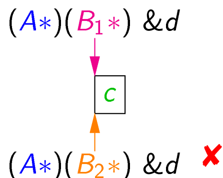
```
struct    A              { };
struct    B1  :  A       { };
struct    B2  :  A       { };
struct    D   :  B1, B2  { char c; };
D d       ;
```



Empty base layout

- Minimal size storage, possibly zero (*empty base optimization*)
- Object identity may require further padding

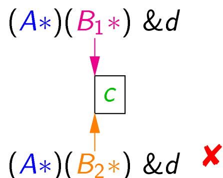
```
struct A        { };
struct B1 : A   { };
struct B2 : A   { };
struct D : B1, B2 { char c; };
D d ;
```

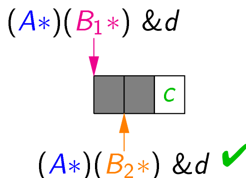


Empty base layout

- Minimal size storage, possibly zero (*empty base optimization*)
- Object identity may require further padding

```
struct A { };
struct B1 : A { };
struct B2 : A { };
struct D : B1, B2 { char c; };
D d ;
```

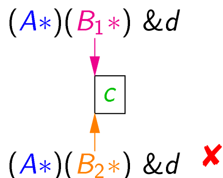


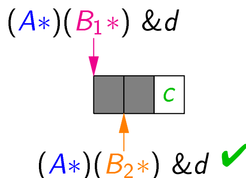


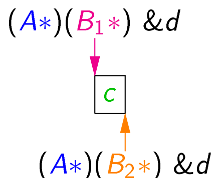
Empty base layout

- Minimal size storage, possibly zero (*empty base optimization*)
- Object identity may require further padding

```
struct A { };
struct B1 : A { };
struct B2 : A { };
struct D : B1, B2 { char c; };
D d ;
```







Empty base layout

- Minimal size storage, possibly zero (*empty base optimization*)
- Object identity may require further padding

$\texttt{struct}\ A\ \{\ \};$

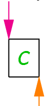
$\texttt{struct}\ B_1 : A\ \{\ \};$

$\texttt{struct}\ B_2 : A\ \{\ \};$

$\texttt{struct}\ D : B_1, B_2\ \{\ \texttt{char}\ c;\ \};$

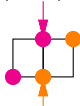
```
D d, t[2];
```

$(A*)(B_1*)\ \&d$



$(A*)(B_2*)\ \&d$

$(A*)(B_1*)\ \&t[1]$



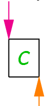
$(A*)(B_2*)\ \&t[0]$ ✘

Empty base layout

- Minimal size storage, possibly zero (*empty base optimization*)
- Object identity may require further padding

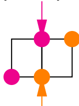
```
struct  A              { };
struct  B1  :  A       { };
struct  B2  :  A       { };
struct  D   :  B1, B2  { char c; };
D d, t[2];
```

$(A*)(B_1*)\ \&d$



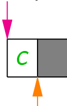
$(A*)(B_2*)\ \&d$

$(A*)(B_1*)\ \&t[1]$



$(A*)(B_2*)\ \&t[0]$ ✘

$(A*)(B_1*)\ \&d$



$(A*)(B_2*)\ \&d$ ✔

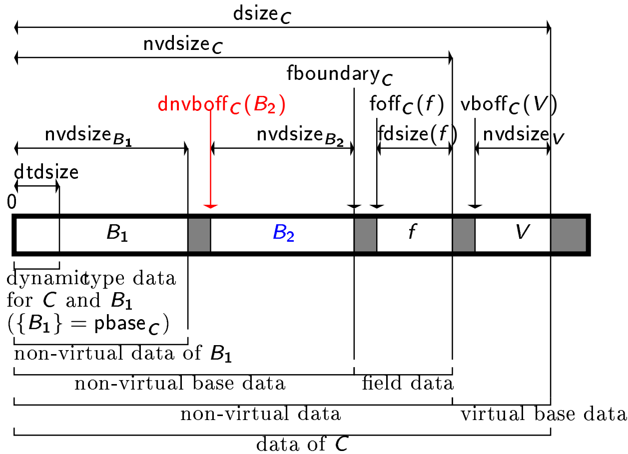
Overview of our work

- A formalization of a family of C++ object layout algorithms
- A formalization of the semantics of C++ objects, with the main interesting features:
  - multiple inheritance
  - virtual inheritance
  - embedded structure fields
- A semantic correctness proof of such layout
- All proofs done with the Coq proof assistant

Outline

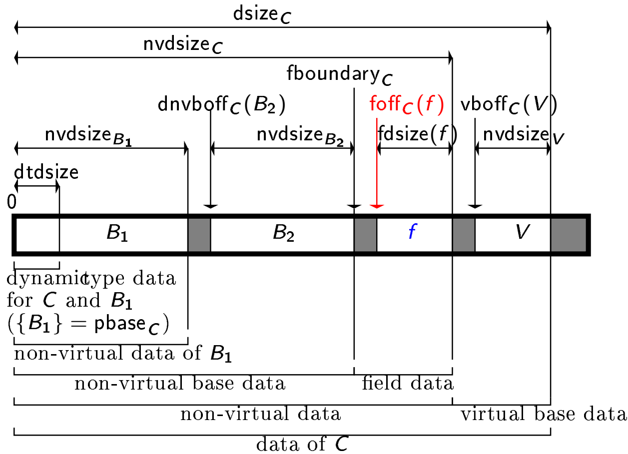
1. Formalization of the layout of C++ classes
2. Formal semantics of C++ object model
3. Real-world layout algorithms
4. Conclusion and perspectives

Layout parameters



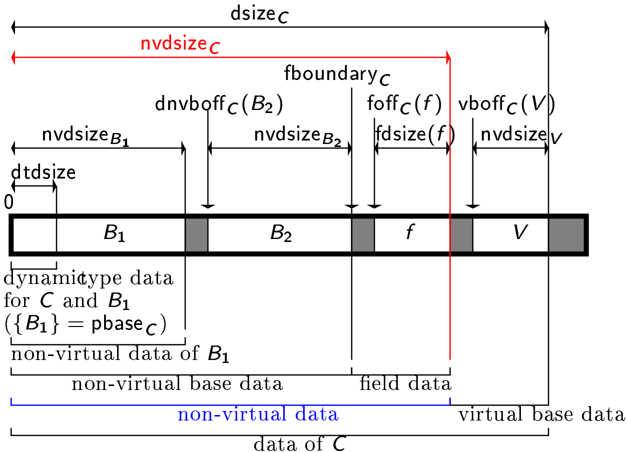
dnvboff$_C(B)$ is the offset, within C, of the direct non-virtual base B of C

Layout parameters



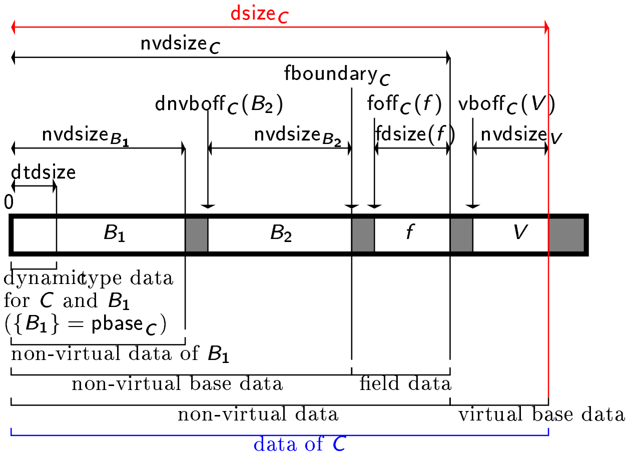
foff$_C(f)$ is the offset, within C, of the field f declared in C

Layout parameters



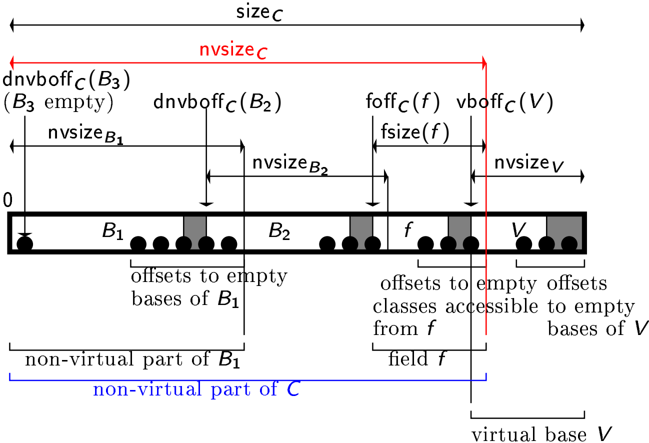
nvdsize$_C$ is the data size of the **non-virtual** part of C, **excluding** its tail padding

Layout parameters



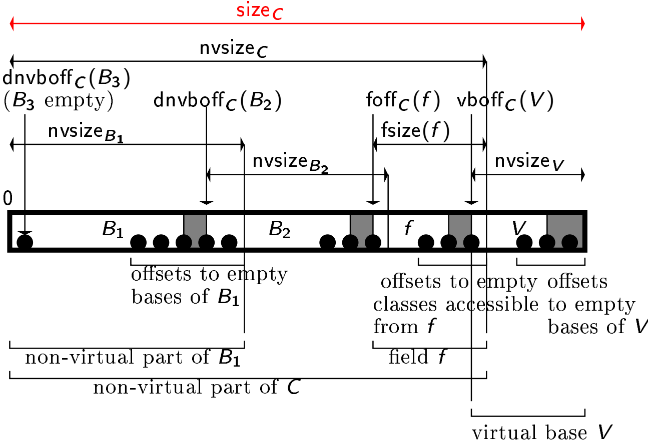
dsize$_C$ is the data size of a full C object, **excluding** its tail padding

Layout parameters



nvsize$_C$ is the total size of the **non-virtual** part of C

Layout parameters



size_C is the total size of a full C object

Sufficient layout conditions

26 conditions deemed sufficient to make a layout semantically correct, among which:

- C2: $\mathrm{foff}_C(F) + \mathrm{fsize}(F) \leq \mathrm{nvsize}_C$
- C9:
$$\begin{array}{rl} & [\mathrm{foff}_C(F_1), \mathrm{foff}_C(F_1) + \mathrm{fdsize}(F_1)) \\ \# & [\mathrm{foff}_C(F_2), \mathrm{foff}_C(F_2) + \mathrm{fdsize}(F_2)) \end{array}$$

Those conditions do not deterministically fix the offsets and sizes, it is up to the algorithm.

Correctness of sufficient layout conditions

Lemma

Those conditions guarantee common sense properties about fields and bases:

- *two different scalar fields are stored in disjoint memory zones*
- *fields are correctly aligned*
- *different subobjects of the same type are stored at different offsets*
- *scalar field writes do not impact dynamic type data*

Correctness of sufficient layout conditions

Proved in Coq

Lemma

Those conditions guarantee common sense properties about fields and bases:

- *two different scalar fields are stored in disjoint memory zones*
- *fields are correctly aligned*
- *different subobjects of the same type are stored at different offsets*
- *scalar field writes do not impact dynamic type data*

Outline

1 Formalization of the layout of C++ classes

2 Formal semantics of C++ object model

3 Real-world layout algorithms

4 Conclusion and perspectives

History of formal semantics of C++ subobjects

- First formalization: Rossie & Friedman, *An algebraic semantics of subobjects* (OOPSLA'95)
- First machine formalization: Wasserrab, Nipkow et al., *An Operational Semantics and Type Safety Proof for Multiple Inheritance in C++* (OOPSLA'06)

Designating subobjects with paths

$nv_{D,B}$	$::=$	$D :: \cdots :: B$	Non-virtual inheritance path
$p_{D,B}$	$::=$	$(\text{Repeated}, nv_{D,B})$	B is a non-virtual base of D
	$\mid$	$(\text{Shared}, nv_{V,B})$	V is a virtual base of D and B is a non-virtual base of V

Designating subobjects with paths

We extended those works to embedded structures and arrays.

$nv_{D,B}$	$::=$	$D :: \cdots :: B$	Non-virtual inheritance path
$p_{D,B}$	$::=$	$(\text{Repeated}, nv_{D,B})$	B is a non-virtual base of D
	$\mid$	$(\text{Shared}, nv_{V,B})$	V is a virtual base of D and B is a non-virtual base of V
$subo$	$::=$	$(idx, p, f)^* \ (idx', p')$	path to a subobject inside an array through embedded structure array fields

A core language

We defined a core language for C++ multiple inheritance, featuring the most interesting object-oriented features:

$Stmt$	$::=$	$var := var\texttt{->}_C f$	Reading scalar field or pointing to structure field
	$\mid$	$var\texttt{->}_C f := var$	Writing scalar field
	$\mid$	$var := \&var[var]_C$	Pointing to array cell
	$\mid$	$var := \texttt{static_cast}\langle A\rangle_C(var)$	Static cast
	$\mid$	$var := \texttt{dynamic_cast}\langle A\rangle_C(var)$	Dynamic cast
	$\mid$	$\ldots$	Structured control

A core language

We defined a core language for C++ multiple inheritance, featuring the most interesting object-oriented features:

$Stmt$	$::=$	$var := var\texttt{->}_C f$	Reading scalar field or pointing to structure field
	$\mid$	$var\texttt{->}_C f := var$	Writing scalar field
	$\mid$	$var := \&var[var]_C$	Pointing to array cell
	$\mid$	$var := \texttt{static_cast}\langle A \rangle_C(var)$	Static cast
	$\mid$	$var := \texttt{dynamic_cast}\langle A \rangle_C(var)$	Dynamic cast
	$\mid$	$\ldots$	Structured control

No function calls: the need for dynamic type data is highlighted by dynamic cast instead of virtual function calls.

Compilation of object-oriented operations

$$[\![x := x'\texttt{->}_C F]\!] = x := \texttt{load}(\mathrm{scsize}_t, x' + \mathrm{foff}_C(F))$$
$$\text{(if } F = (f, t) \text{ is a scalar field of } C\text{)}$$

$$[\![x\texttt{->}_C F := x']\!] = \texttt{store}(\mathrm{scsize}_t, x + \mathrm{foff}_C(F), x')$$
$$\text{(if } F = (f, t) \text{ is a scalar field of } C\text{)}$$

$$[\![x := x'\texttt{->}_C F]\!] = x := x' + \mathrm{foff}_C(F)$$
$$\text{(if } F \text{ is a structure array field of } C\text{)}$$

$$[\![x := \&x_1[x_2]_C]\!] = x := x_1 + \mathrm{size}_C \times x_2$$

$$[\![x := x_1 \texttt{ == } x_2]\!] = x := x_1 \texttt{ == } x_2$$

Compilation of casts

- For static casts, there are two cases:
  - For a non-virtual subobject $p_{D,B} = (\text{Repeated}, l)$:

$$[\![x := \texttt{static_cast}\langle B\rangle_D(x')]\!] = x := x' + \text{nvsoff}(l)$$
$$[\![x := \texttt{static_cast}\langle D\rangle_B(x')]\!] = x := x' - \text{nvsoff}(l)$$

  - For a subobject through virtual inheritance $p_{D,B} = (\text{Shared}, V :: l)$, the offset of the virtual base V of C must be looked up in the dynamic type data:

$$[\![x := \texttt{static_cast}\langle A\rangle_C(x')]\!] =$$
$$t := \texttt{load}(\text{dtdsize}, x'); x := x' + \texttt{read_vboff}(t, V) + \text{nvsoff}(l)$$

(reads through dynamic type data are left abstract)

Compilation of casts

- For static casts, there are two cases:
  - For a non-virtual subobject $p_{D,B} = (\text{Repeated}, l)$:

$$[\![x := \texttt{static_cast}\langle B\rangle_D(x')]\!] = x := x' + \text{nvsoff}(l)$$
$$[\![x := \texttt{static_cast}\langle D\rangle_B(x')]\!] = x := x' - \text{nvsoff}(l)$$

  - For a subobject through virtual inheritance $p_{D,B} = (\text{Shared}, V :: l)$, the offset of the virtual base V of C must be looked up in the dynamic type data:

$$[\![x := \texttt{static_cast}\langle A\rangle_C(x')]\!] =$$
$$t := \texttt{load}(\text{dtdsize}, x'); x := x' + \texttt{read_vboff}(t, V) + \text{nvsoff}(l)$$

  (reads through dynamic type data are left abstract)

- Dynamic cast is compiled as a read through the pointer to dynamic type data

Semantics preservation

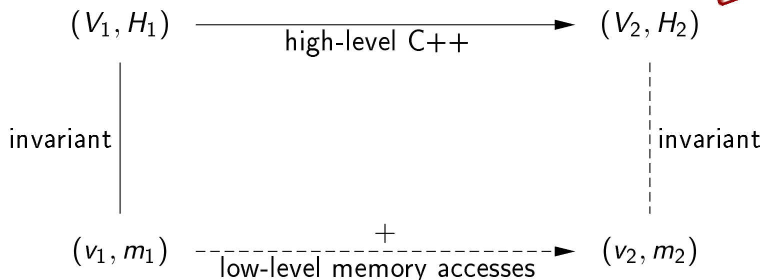
Theorem

If the layout algorithm satisfies the 26 sufficient conditions, then the compilation scheme preserves the semantics of programs.

Semantics preservation

Theorem

If the layout algorithm satisfies the 26 sufficient conditions, then the compilation scheme preserves the semantics of programs.



Outline

1. Formalization of the layout of C++ classes
2. Formal semantics of C++ object model
3. Real-world layout algorithms
4. Conclusion and perspectives

Common vendor ABI layout algorithm

- Application Binary Interface: agreement on data layout for programs compiled by different compilers for the same platform
- Common vendor ABI designed by a consortium of compiler designers, `http://www.codesourcery.com/public/cxx-abi/`
- Initially for Itanium, then adopted by GNU GCC and almost all compiler builders and platforms (except Microsoft)
- A fairly complicated algorithm, difficult to implement

Common vendor ABI layout algorithm

- [C++FDIS] The **Final Draft International Standard, Programming Language C++**, ISO/IEC FDIS 14882:1998(E). References herein to the "C++ Standard," or to just the "Standard," are to this document.

Chapter 2: Data Layout

2.1 General

In what follows, we define the memory layout for C++ data objects. Specifically, for each type, we specify the following information about an object O of that type:

- the *size* of an object, *sizeof*(O);
- the *alignment* of an object, *align*(O); and
- the *offset* within O, *offset*(C), of each data component C, i.e. base or member.

For purposes internal to the specification, we also specify:

- *dsize*(O): the *data size* of an object, which is the size of O without tail padding.
- *nvsize*(O): the *non-virtual size* of an object, which is the size of O without virtual bases.
- *nvalign*(O): the *non-virtual alignment* of an object, which is the alignment of O without virtual bases.

2.2 POD Data Types

The size and alignment of a type which is a POD for the purpose of layout is as specified by the base (C) ABI. Type bool has size and alignment 1. All of these types have data size and non-virtual size equal to their size. (We ignore tail padding for PODs because the Standard does not allow us to use it for anything else.)

2.3 Member Pointers

A pointer to data member is an offset from the base address of the class object containing it, represented as a **ptrdiff_t**. It has the size and alignment attributes of a **ptrdiff_t**. A NULL pointer is represented as -1.

A pointer to member function is a pair as follows:

ptr:

For a non-virtual function, this field is a simple function pointer. (Under current base Itanium psABI conventions, that is a pointer to a GP/function address pair.) For a virtual function, it is 1 plus the virtual table offset (in bytes) of the function, represented as a **ptrdiff_t**. The value zero represents a NULL pointer, independent of the adjustment field value below.

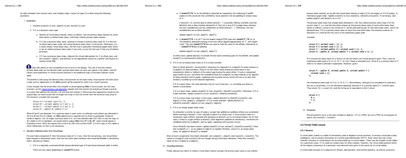
adj:

The required adjustment to *this*, represented as a **ptrdiff_t**.

It has the size, data size, and alignment of a class containing those two members, in that order. (For 64-bit Itanium, that will be 16, 16, and 8 bytes respectively.)

2.4 Non-POD Class Types

For a class type C which is not a POD for the purpose of layout, assume that all component types (i.e. proper base classes and non-static data member types) have been laid out, defining size, data size, non-virtual size, alignment, and non-virtual alignment. (See the description of these terms in **General** above.) Further, assume



Correctness of the common vendor ABI layout algorithm

Theorem

This algorithm satisfies the 26 sufficient conditions for the correctness of object layout. Thus, it can be fed to the compiler to obtain a verified compiler preserving the semantics of programs.

Proved in Coq

Object layout entirely proved except a controversial optimization on *virtual primary bases*.

Extension: an optimized layout algorithm

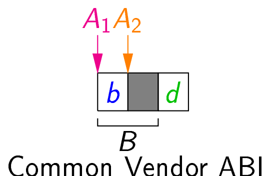
We developed an extension of this algorithm to allow further reusing of the tail paddings of non-virtual bases and fields.

struct	Z	:		{ };
struct	A_1	:	Z	{ };
struct	A_2	:	Z	{ };
struct	B	:	A_1, A_2	{ char b };
struct	D	:	B	{ char d };

Extension: an optimized layout algorithm

We developed an extension of this algorithm to allow further reusing of the tail paddings of non-virtual bases and fields.

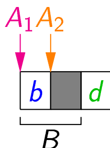
`struct`	Z	:		`{ };`
`struct`	A_1	:	Z	`{ };`
`struct`	A_2	:	Z	`{ };`
`struct`	B	:	A_1, A_2	`{ char` b `};`
`struct`	D	:	B	`{ char` d `};`



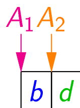
Extension: an optimized layout algorithm

We developed an extension of this algorithm to allow further reusing of the tail paddings of non-virtual bases and fields.

```
struct Z  :        { };
struct A1 : Z      { };
struct A2 : Z      { };
struct B  : A1, A2 { char b };
struct D  : B      { char d };
```



Common Vendor ABI



Our optimizations

Extension: an optimized layout algorithm

Proved in Coq

Theorem

This extended algorithm satisfies the 26 sufficient conditions for the correctness of object layout. Thus, it can be fed to the compiler to obtain a verified compiler preserving the semantics of programs.

The proof is simpler as the algorithm behaves very closely to the conditions.

Outline

1. Formalization of the layout of C++ classes
2. Formal semantics of C++ object model
3. Real-world layout algorithms
4. Conclusion and perspectives

Summary

- A general formal model for C++ object-oriented features
- First machine-checked formalization of sufficient conditions for layout parameters guaranteeing semantics preservation
- First machine-checked correctness proof of a commonly used layout algorithm
- Suggests and justifies further optimizations
- Positive feedback from C++ Standard Committee

Ongoing and future work

Ongoing work:

- Even better layout algorithms (bidirectional, etc.)
- Semantics and compilation of construction and destruction

Future work:

- Concrete representation of virtual tables
- Dream of a verified C++ compiler in the style of CompCert

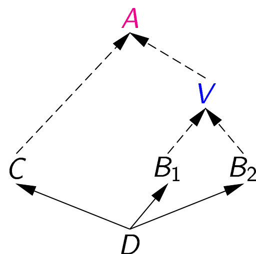
Thank you for your attention

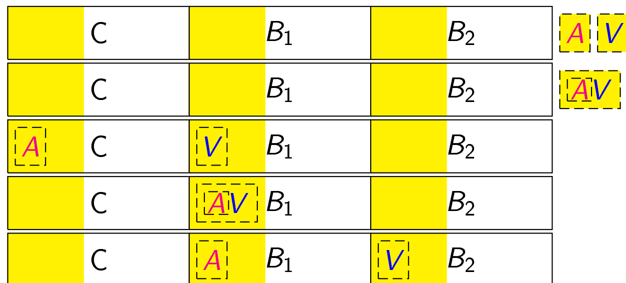
- Coq development fully available on the Web: `http://gallium.inria.fr/~tramanan/cxx/object-layout`
- For further information: `Tahina.Ramananandro@inria.fr`

Figure: Field layout in Texas

Virtual primary bases

`struct`	A			`{ virtual void f(); };`
`struct`	V	:	`virtual` A	`{ };`
`struct`	C	:	`virtual` A	`{ };`
`struct`	B_1	:	`virtual` V	`{ };`
`struct`	B_2	:	`virtual` V	`{ };`
`struct`	D	:	C, B_1, B_2	`{ };`





Back

Buggy EBO implementations

- MetroWerks CodeWarrior C++ 4.0 and IBM too agressive, fail to enforce object identity (`http://www.cantrip.org/emptyopt.html`)
- Excerpt from `http://bytes.com/topic/c/answers/129536-multiple-inheritance-size-problem`):

```
struct Empty           {};
struct Derived: Empty {
 Empty value;
};
Derived d;
```

Microsoft Visual C++ 7.1 and Borland C++ Builder 5.x erroneously give `((Empty*) &d) == &d.value`

Unfaithful implementations of Common Vendor ABI

Excerpt from Mark Mitchell,
http://gcc.gnu.org/ml/gcc/2002-08/msg01640.html

```
struct A { virtual void f(); char c1; };
struct B { B(); char c2; };
struct C : public A, public virtual B {};
```

GNU GCC 3.2 does not reuse the alignment tail padding of A for B as required by the ABI.

Sizes

(C1) $\text{dnvboff}_C(B) + \text{nvsize}_B \leq \text{nvsize}_C$ if B direct non-virtual base of C

(C2) $\text{foff}_C(f) + \text{fsize}(f) \leq \text{nvsize}_C$ if f field of C

(C3) $\text{vboff}_C(B) + \text{nvsize}_B \leq \text{size}_C$ if B generalized virtual base of C

(C4) $\text{dsize}_C \leq \text{size}_C$

(C5) $0 < \text{nvsize}_C$

Field separation

(C6) $[\mathrm{dnvboff}_C(B_1), \mathrm{dnvboff}_C(B_1) + \mathrm{nvdsize}_{B_1})$
$\quad \# \ [\mathrm{dnvboff}_C(B_2), \mathrm{dnvboff}_C(B_2) + \mathrm{nvdsize}_{B_2})$
if B_1, B_2 distinct non-empty non-virtual direct bases of C

(C7) $\mathrm{dnvboff}_C(B) + \mathrm{nvdsize}_B \leq \mathrm{fboundary}_C$
if B non-empty non-virtual direct base of C

(C8) $\mathrm{fboundary}_C \leq \mathrm{foff}_C(f)$ if f relevant field of C

(C9) $[\mathrm{foff}_C(f_1), \mathrm{foff}_C(f_1) + \mathrm{fdsize}(f_1))$
$\quad \# \ [\mathrm{foff}_C(f_2), \mathrm{foff}_C(f_2) + \mathrm{fdsize}(f_2))$
if f_1 and f_2 are distinct relevant fields of C

(C10) $\mathrm{foff}_C(f) + \mathrm{fdsize}(f) \leq \mathrm{nvdsize}_C$ if f relevant field of C

(C11) $\mathrm{fboundary}_C \leq \mathrm{nvdsize}_C$

(C12) $[\mathrm{vboff}_C(B_1), \mathrm{vboff}_C(B_1) + \mathrm{nvdsize}_{B_1})$
$\quad \# \ [\mathrm{vboff}_C(B_2), \mathrm{vboff}_C(B_2) + \mathrm{nvdsize}_{B_2})$
if B_1, B_2 distinct non-empty generalized virtual bases of C

(C13) $\mathrm{vboff}_C(B) + \mathrm{nvdsize}_B \leq \mathrm{dsize}_C$

Field alignment – Dynamic type data

Field alignment

(C14) $(\mathrm{falign}(f) \mid \mathrm{foff}_C(f))$ and $(\mathrm{falign}(f) \mid \mathrm{nvalign}_C)$
if f field of C

(C15) $(\mathrm{nvalign}_B \mid \mathrm{dnvboff}_C(B))$ and $(\mathrm{nvalign}_B \mid \mathrm{nvalign}_C)$
if B non-virtual base of C

(C16) $(\mathrm{dtdalign} \mid \mathrm{nvalign}_C)$ if C is dynamic

(C17) $(\mathrm{nvalign}_B \mid \mathrm{vboff}_C(B))$ and $(\mathrm{nvalign}_B \mid \mathrm{align}_C)$
if B is a generalized virtual base of C

(C18) $(\mathrm{align}_C \mid \mathrm{size}_C)$

Dynamic type data

(C19) $\mathrm{dtdsize} \leq \mathrm{fboundary}_C$

(C20) $\mathrm{pbase}_C = \varnothing \;\Rightarrow\; \mathrm{dtdsize} \leq \mathrm{dnvboff}_C(B)$
if B is a non-empty non-virtual direct base of C

(C21) $\mathrm{pbase}_C = \{B\} \;\Rightarrow\; \mathrm{dnvboff}_C(B) = 0$

Identity of subobjects

$$
\begin{aligned}
\mathsf{eboffs}_C &=_{\mathsf{def}} \bigcup_{B \in \mathsf{vbases}_C \cup \{C\}} \mathsf{vboff}_C(B) + \mathsf{neboffs}_B \\
\mathsf{nveboffs}_C &=_{\mathsf{def}} \text{if } C \text{ is empty then } \{(C, 0)\} \text{ else } \varnothing \\
&\cup \bigcup_{B \in \mathsf{dnvbases}_C} \mathsf{dnvboffs}_C(B) + \mathsf{nveboffs}_B \\
&\cup \bigcup_{\substack{(f, B, n) \in \mathsf{stfields}_C \\ 0 \le i < n}} \mathsf{foff}_C(f, B, n) + i \cdot \mathsf{size}_B + \mathsf{eboffs}_B
\end{aligned}
$$

(C22) C non-empty $\Rightarrow 0 < \mathsf{nvdsize}_C$

(C23) $(\mathsf{dnvboff}_C(B_1) + \mathsf{nveboffs}_{B_1}) \# (\mathsf{dnvboff}_C(B_2) + \mathsf{nveboffs}_{B_2})$

if B_1, B_2 distinct non-virtual bases of C

(C24) $(\mathsf{dnvboff}_C(B_1) + \mathsf{nveboffs}_{B_1}) \# \bigcup_{0 \le j < n} \mathsf{foff}_C(f, B_2, n) + j \cdot \mathsf{size}_{B_2} + \mathsf{eboffs}_{B_2}$

if B_1 non-virtual base of C and (f, B_2, n) structure field of C

(C25) $\bigcup_{0 \le j_1 < n_1} \mathsf{foff}_C(f_1, B_1, n_1) + j_1 \cdot \mathsf{size}_{B_1} + \mathsf{eboffs}_{B_1} \# \bigcup_{0 \le j_2 < n_2} \mathsf{foff}_C(f_2, B_2, n_2) + j_2 \cdot \mathsf{size}_{B_2} + \mathsf{eboffs}_{B_2}$

if (f_1, B_1, n_1) and (f_2, B_2, n_2) distinct structure fields of C

(C26) $(\mathsf{vboff}_C(B_1) + \mathsf{nveboffs}_{B_1}) \# (\mathsf{vboff}_C(B_2) + \mathsf{nveboffs}_{B_2})$

if B_1, B_2 distinct generalized virtual bases of C

Thank you for your attention

`Tahina.Ramananandro@inria.fr`
`http://gallium.inria.fr/~tramanan/cxx/object-layout`

1. Formalization of the layout of C++ classes
2. Formal semantics of C++ object model
3. Real-world layout algorithms
4. Conclusion and perspectives