Mechanized Formal Semantics and Verified Compilation for C++ objects

Tahina Ramananandro[1]

[1] INRIA Paris-Rocquencourt

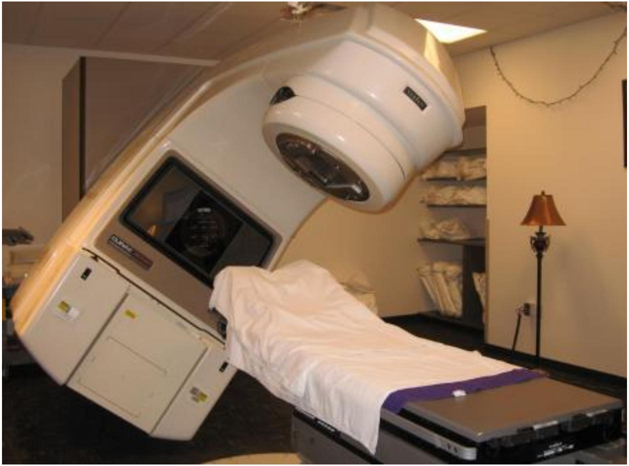
January 10th, 2012

Software errors

- Software is ubiquitous
- Software errors (bugs, failures) mostly with limited effects. . .

Software errors

- Software is ubiquitous
- Software errors (bugs, failures) mostly with limited effects. . .
- . . . except in specific areas of **critical software**, where the slightest bug can lead to dramatic consequences:
  - medical devices
  - transportation (space, avionics, railways)
  - military applications



Therac 25 radiotherapy machine (1985): at least 6 patients dead due to software activating wrong radiation mode


Photo Service Optique - CSG - © ESA - CNES
ARIANE 501 - 4 Juin 1996

→

Ariane 5 maiden flight (1996): US$370 million lost material and project delayed by 4 years due to overflow in floating-point computations

Trusted software

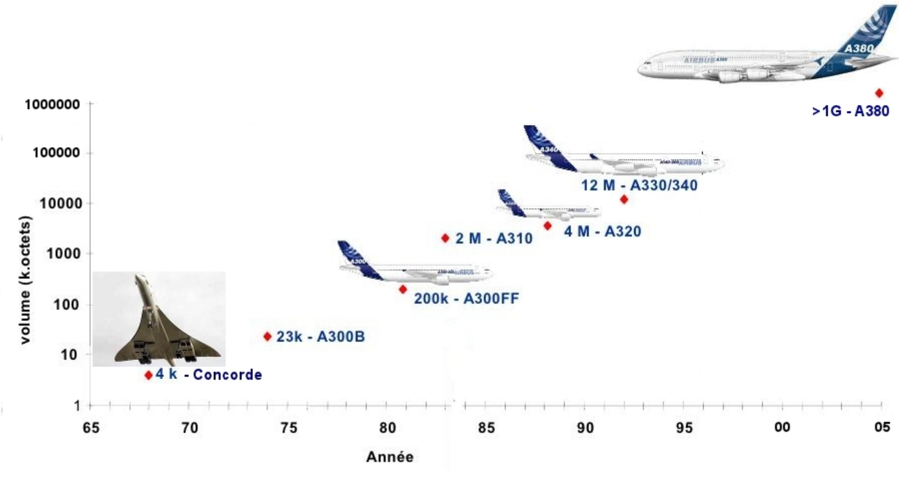
- Software more and more present in critical systems
- Need highest quality
- Need to be trusted

Software testing

Usual approach in industry: Testing and manual code reviews

- Required in avionics by DO-178B official regulations
- Software errors caused no casualties so far in avionics
- All cases covered?
- Costs?

Scalability of software testing?



Formal verification of software

A complementary approach: software verification by formal methods:

model-checking, abstract interpretation, deductive verification, automated program generation. . .

Formal verification of software

A complementary approach: software verification by formal methods:

model-checking, abstract interpretation, deductive verification, automated program generation...

- Stronger guarantees
- Exhaustive: all behaviours taken into account
- No need to run the software
- Solid mathematical backgrounds

Formal verification of software

Program

```
int main () {
  int x = 21;
  return x+x;
}
```

Specification

- The program terminates
- The program is not interrupted by an error
- If the program terminates, then it returns 42

Formal verification of software

Program

```
int main () {
  int x = 21;
  return x+x;
}
```

$\rightsquigarrow$
meets
?

Specification

- The program terminates
- The program is not interrupted by an error
- If the program terminates, then it returns 42

Mechanized Formal Semantics. . .

Reasoning on a program needs studying its meaning, thus knowing about its language.

- Need a mathematical description of the programming language: its *semantics*

Mechanized Formal Semantics. . .

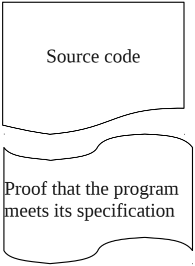
Reasoning on a program needs studying its meaning, thus knowing about its language.

- Need a mathematical description of the programming language: its *semantics*
- As opposed to language definitions in practice: textual standardization documents or even reference implementations (compilers, interpreters)
    - prone to ambiguities, forgotten undefined cases,. . .

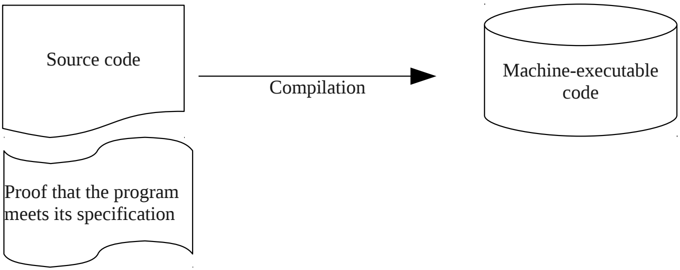
Mechanized Formal Semantics. . .

- Formal verification based on (semi-)automated computer tools: verification condition generators, theorem provers, **proof assistants (e.g. Coq)**. . . .
- Thus, desirable to formalize language semantics inside such mechanical systems.

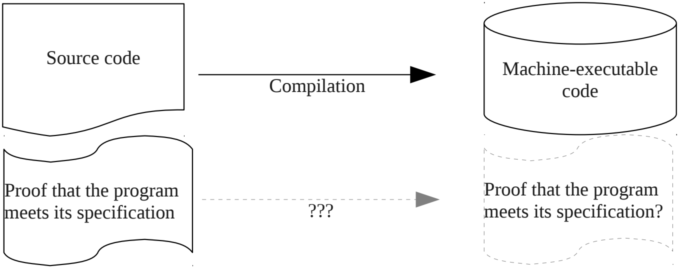
. . . and Verified Compilation . . .



. . . and Verified Compilation . . .



. . . and Verified Compilation . . .



Verified compilation by Semantics preservation
relies on the formal semantics
of (both) languages

. . . for C++ Objects

- C++ is one of the most used languages in the world
  - In everyday life: Firefox, Thunderbird, Photoshop, HotSpot JVM,. . .
  - More and more used in critical embedded software: Lockheed Martin, Mars Rover. . .

...for C++ Objects

- C++ is one of the most used languages in the world
  - In everyday life: Firefox, Thunderbird, Photoshop, HotSpot JVM,...
  - More and more used in critical embedded software: Lockheed Martin, Mars Rover...
- More functionalities for more abstraction power than C or assembly languages
  - **object-oriented programming**
  - generic programming (templates), exceptions,...
- But C++ semantics allegedly complicated
  - defined by textual standard ($>$ 1000 pages) ISO/IEC 14882:2011

...for C++ Objects

- C++ is one of the most used languages in the world
  - In everyday life: Firefox, Thunderbird, Photoshop, HotSpot JVM,...
  - More and more used in critical embedded software: Lockheed Martin, Mars Rover...
- More functionalities for more abstraction power than C or assembly languages
  - **object-oriented programming**
  - generic programming (templates), exceptions,...
- But C++ semantics allegedly complicated
  - defined by textual standard ($>$ 1000 pages) ISO/IEC 14882:2011

A formal semantics of C++ is needed.

We focus on the C++ object model (multiple inheritance, construction and destruction).

Thesis:
The semantics and compilation of the C++ object model can be formally trusted.

Outline

1. Overview of the C++ object model
2. Formal semantics
3. Verified compilation
4. Conclusion and perspectives

Outline

Initializing objects using a constructor

```
struct Point {
  double x;
  double y;
  Point (double x0, double y0): x(x0), y(y0) {}
};

main () {
  Point c = Point (1.2, 3.4);
}
```

Initializing embedded objects

```
struct Point {
  double x;
  double y;
  Point (double x0, double y0): x(x0), y(y0) {}
};

struct Segment {
  Point p1;
  Point p2;
  Segment (double x1, double y1, double x2, double y2):
    p1 (x1, y1), p2 (x2, y2) {}
};

main () {
  Segment s = Segment (1.2, 3.4, 18.42, 17.29);
}
```

Outline

Object destruction

```
main() {
   File f = File("toto.txt");
   f.write("Hello world!");
}
```

Object destruction

```
struct File {
   FILE* handle;
   void write(char* string)  ...

   // Constructor
   File(char* name):  handle(fopen(name, "w")) {}



}

main() {
   File f = File("toto.txt");
   f.write("Hello world!");
}
```

Object destruction

```
struct File {
   FILE* handle;
   void write(char* string)  ...

   // Constructor
   File(char* name):  handle(fopen(name, "w")) {}

   // Destructor
   ~File()                    { fclose(handle); }
}

main() {
   File f = File("toto.txt");
   f.write("Hello world!");
} // automatic destructor call on scope exit
  // Resource acquisition is initialization (RAII)
```

Destructing embedded objects

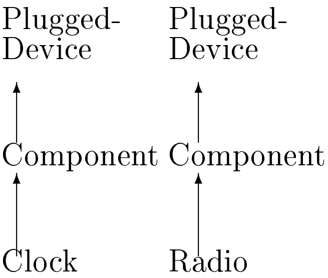
```
struct LockFile {
  Lock lock;
  File file;
  LockFile (char* name): lock (), file (name) {}
};
```

Two subobjects of the same object must be destructed in the reverse order of their construction.

Outline

1. Overview of the C++ object model
   - Construction: object initialization
   - Destruction: resource management
   - C++ multiple inheritance
   - Overview of our work
2. Formal semantics
3. Verified compilation
4. Conclusion and perspectives

Single inheritance



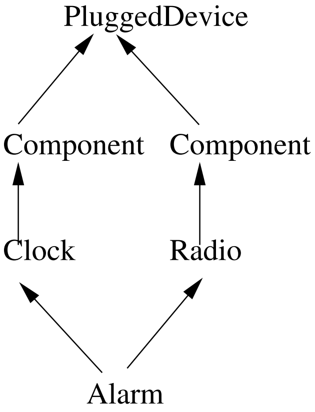

```
struct PluggedDevice {
  int plug;
};

struct Component: PluggedDevice {
  int switch;
};

struct Clock: Component {
  int time;
};

struct Radio: Component {
  int volume;
};
```

Two kinds of multiple inheritance




```
struct PluggedDevice {
  int plug;
};

struct Component:
virtual PluggedDevice {
  int switch;
};

struct Clock: Component {
  int time;
};
struct Radio: Component {
  int volume;
};

struct Alarm: Clock, Radio {
  int alarmTime;
};
```

Outline

Overview of our work

- A formalization of the semantics of C++ objects, with the main interesting features:
  - multiple inheritance
  - virtual inheritance
  - embedded structure fields
  - static and dynamic casts, virtual function calls
  - object construction and destruction
- Properties of object construction and destruction
- A verified compiler to a Cminor-style 3-address language with low-level memory accesses

Overview of our work

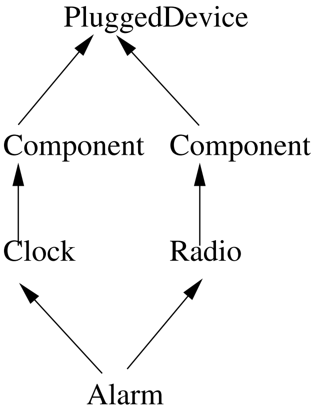
- A formalization of the semantics of C++ objects, with the main interesting features:
  - multiple inheritance
  - virtual inheritance
  - embedded stru
  - static and dyna s
  - object construct
- Properties of object c estruction
- A verified compiler to or-style 3-address language with low-level memory accesses

Outline

Outline

1. Overview of the C++ object model
2. Formal semantics
   - C++ multiple inheritance
   - Object construction and destruction
3. Verified compilation
4. Conclusion and perspectives

The algebra of subobjects



- From Alarm to Component :
  - Alarm :: Clock :: Component :: nil
  - Alarm :: Radio :: Component :: nil
  - Alarm :: Component :: nil
- From Alarm to PluggedDevice :
  - PluggedDevice :: nil

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
$subo$	$::=$	$(idx, p, f) \ldots \quad (idx', p')$	path to a subobject inside an array thru embedded structure array fields

A core language

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

$Stmt$	$::=$	$var := var\text{->}_C f$	Reading scalar field or pointing to structure field
	$\mid$	$var\text{->}_C f := var$	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
	$\mid$	$var := var\text{->}_C f(var, \dots)$	Virtual function call
	$\mid$	$\{C\,var[n] = \{Init_C, \dots\}; Stmt\}$	Block-scoped object
	$\mid$	$\dots$	Structured control
$Init_C$	$::=$	$Stmt; C(var, \dots)$	Initializer

A core language

$$
\begin{array}{rcll}
\mathit{Funct} & ::= & \texttt{virtual}\ f(var, \ldots)\{\mathit{Stmt}\} & \text{Virtual function} \\
\mathit{Finit}_m & ::= & & \text{Data member} \\
 & & & \text{initializers} \\
 & & m\{\mathit{Init}_A \ldots\} & \text{Structure} \\
 & | & m(\mathit{Stmt}, var) & \text{Scalar} \\
 & & & \\
\mathit{Constr}_C & ::= & C(var, \ldots) : \mathit{Init}_{B1}, \ldots, \mathit{Init}_{V1}, \ldots, & \text{Constructor} \\
\mathit{Destr}_C & ::= & \tilde{}C()\{\mathit{Stmt}\} & \text{Destructor} \\
 & & \mathit{Finit}_m, \ldots \{\mathit{Stmt}\} & \\
 & & & \\
\mathit{Class} & ::= & \texttt{struct}\ C : B1, \ldots, \texttt{virtual}\ V1, \ldots & \text{Class definition} \\
 & & \{\mathit{Constr}_C \ldots; \mathit{Funct} \ldots; \mathit{Destr}_C\} & \\
\mathit{Prog} & ::= & \mathit{Class} \ldots & \text{Program}
\end{array}
$$

Outline

The semantics of object construction and destruction

We have designed a small-step operational semantics to precisely model the different steps of object construction and destruction.

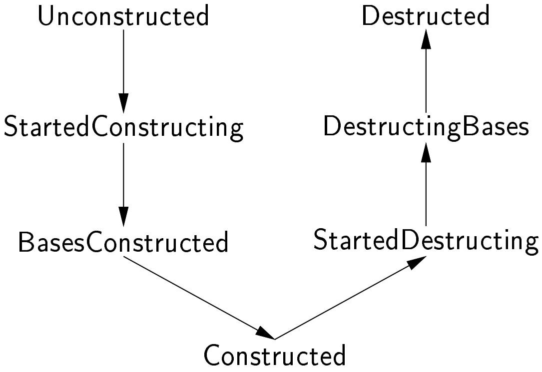
The semantics of object construction and destruction

We have designed a small-step operational semantics to precisely model the different steps of object construction and destruction.

- Resolution of virtual function calls
- Construction and destruction protocol
- Guarantees during construction and destruction

The construction states of a subobject

Each (inheritance and/or embedded structure) subobject is equipped at run-time with a *construction state*:



Evolution of the construction state during construction

```
struct C : B {
   int i;

   C ():
     B (),
     i(18)
     {...}
};
```

Unconstructed

Evolution of the construction state during construction

```
struct C : B {
  int i;

  C ():
    B (),
    i(18)
    {...}
};
```

StartedConstructing

Evolution of the construction state during construction

```
struct C : B {
   int i;

   C ():
     B (),
     i(18)
     {...}
};
```

BasesConstructed, virtual functions allowed here

Evolution of the construction state during construction

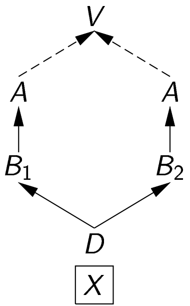
```
struct C : B {
  int i;

  C ():
    B (),
    i(18)
    {...}
};
```

Constructed

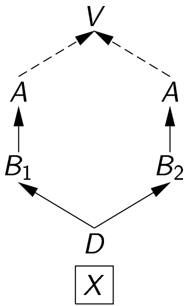
The *lifetime* of a subobject is the set of all states where the construction state of the object is Constructed.

Subobject construction order

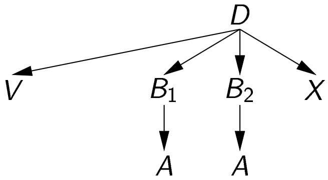


Class hierarchy

Subobject construction order



Class hierarchy



Construction tree

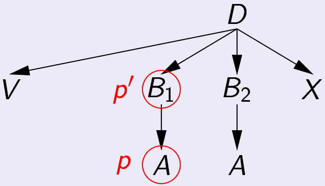
Run-time invariant

To reason about the semantics, we have to specify and prove a run-time invariant. (13000 loc, 2 hours checking time)

Lemma (Parent and child construction states)

If p is a child of p' in the construction tree, then the following table relates their construction states:

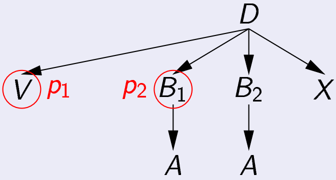
If p' is…	*Then p is…*
Unconstructed	Unconstructed
StartedConstructing	Unconstructed *if p is a field subobject of p'* *between* Unconstructed *and* Constructed *otherwise*
BasesConstructed	Constructed *if p is a base subobject of p'* *between* Unconstructed *and* Constructed *otherwise*
Constructed	Constructed
StartedDestructing	Constructed *if p is a base subobject of p'* *between* Constructed *and* Destructed *otherwise*
DestructingBases	Destructed *if p is a field subobject of p'* *between* Constructed *and* Destructed *otherwise*
Destructed	Destructed



Lemma (Sibling construction states)

Let p_1, p_2 two sibling subobjects such that p_1 appears before p_2 in the construction tree. Then, the following table relates their construction states:

If p_1 is...	*Then p_2 is...*
Unconstructed StartedConstructing BasesConstructed	Unconstructed
Constructed	*in an arbitrary state*
StartedDestructing DestructingBases Destructed	Destructed



Resource management: RAII

Theorem

Each object is constructed and destructed exactly once, in this order.

Theorem

If an object is constructed, then all its subobjects are constructed.

Theorem

If an object is deallocated, then it and all its subobjects are previously constructed, then destructed, in this order.

Theorem

Two subobjects of the same allocated object are destructed in the reverse order of their construction.

Virtual functions during construction and destruction

```
struct B {
  virtual void f () {...}
  B () {
    this->f (); // always calls B::f()
  }
};

struct C : B {
  virtual void f () {...}
  C () : B () {
    this->f (); // always calls C::f()
  }
};
```

Virtual functions during construction and destruction

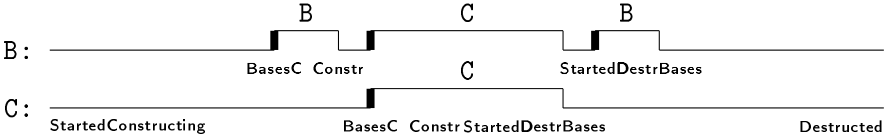
```
struct B {
  virtual void f () {...}
  B () {
    this->f (); // always calls B::f()
  }
};

struct C : B {
  virtual void f () {...}
  C () : B () {
    this->f (); // always calls C::f()
  }
};
```

Safety and modularity

The generalized dynamic type of a subobject

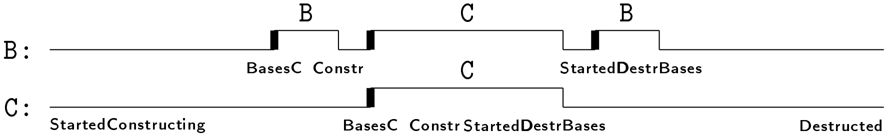
```
struct C : B { ...  };
```



- Considered as the most-derived object for polymorphic operations (dynamic cast, virtual function call)
- In practice, object whose body of constructor/destructor is running

The generalized dynamic type of a subobject

```
struct C : B { ... };
```



- Considered as the most-derived object for polymorphic operations (dynamic cast, virtual function call)
- In practice, object whose body of constructor/destructor is running
- Thick transitions show the times when the compiler must update the pointers to virtual tables

Outline

Compilation passes

Constructors Destructors	$\rightarrow$	Set dynamic type	$\rightarrow$	Low-level memory accesses

Outline

Compilation of object constructors and destructors

Constructors
Destructors $\rightarrow$ Set dynamic type $\rightarrow$ Low-level memory accesses

Compilation of object constructors and destructors

```
struct V {
  V() { ...  }
};

struct B: virtual V {
  B():  V() { ...  }
};

struct D: B {
  int i;
  D():  V(), B(), i(18) {
    printf("Dconstrbody");
  }
};
```

Compilation of object constructors and destructors

```
struct V {
  V() { ... }
};

struct B: virtual V {
  B(): V() { ... }
};

struct D: B {
  int i;
  D(): V(), B(), i(18) {
    printf("Dconstrbody");
  }
};
```

$\longrightarrow$

```
void _constr_D(bool isMostDerived, D* this) {
  if(isMostDerived) {
    _constr_V(false, (V*) this);
  }
  _constr_B(false, (B*) this);
  set dynamic type to D;
  i = 18;
  printf("Dconstrbody");
};
```

Compilation of object constructors and destructors

```
struct V {
   V() { ...  }
};

struct B: virtual V {
   B():  V() { ...  }
};

struct D: B {
   int i;
   D():  V(), B(), i(18) {
     printf("Dconstrbody");
   }
};

main () {
   D d = D();
   ...
   return 42;
}
```

→

```
 void _constr_D(bool isMostDerived, D* this) {
   if(isMostDerived) {
     _constr_V(false, (V*) this);
   }
   _constr_B(false, (B*) this);
   set dynamic type to D;
   i = 18;
   printf("Dconstrbody");
};

main () {
   D d;
   _constr_D(true, &d);
   ...
   _destr_D(true, &d);
   return 42;
}
```

Outline

Representing C++ objects in concrete memory

Constructors Destructors	$\rightarrow$	Set dynamic type	$\rightarrow$	Low-level memory accesses

Representing homogeneous data in memory

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

0	4	8	12
11	22	33	

Representing homogeneous data in memory

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

0	4	8	12
11	22	33	

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

0			12			24
11	22	33	44	55	66	

Representing homogeneous data in memory

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

0	4	8	12
11	22	33	

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

0			12			24
11	22	33	44	55	66	

0		8		16		24
11	44	22	55	33	66	

Representing heterogeneous data in memory

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

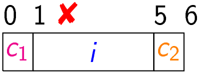
- A naive representation:

0	1	5	6
c_1	i	c_2	

Representing heterogeneous data in memory

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

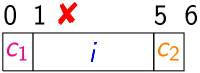
- A naive representation:



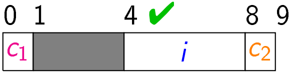
Representing heterogeneous data in memory

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

- A naive representation:



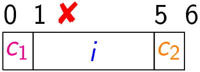
- Correct field alignment requires padding:



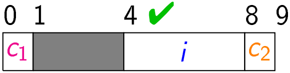
Representing heterogeneous data in memory

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

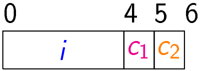
- A naive representation:



- Correct field alignment requires padding:



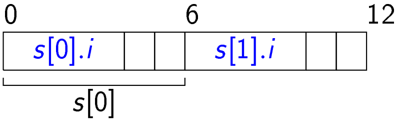
- Reorder fields to save space:



Representing heterogeneous data in memory

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

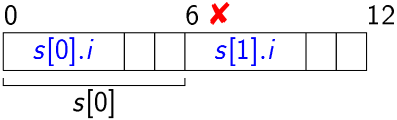
- Making arrays of structures:



Representing heterogeneous data in memory

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

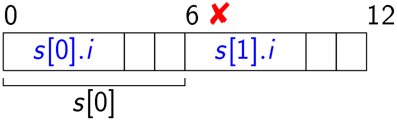
- Making arrays of structures:



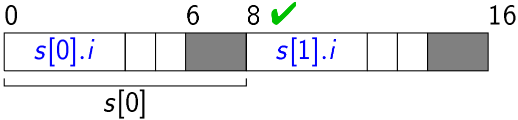
Representing heterogeneous data in memory

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

- Making arrays of structures:



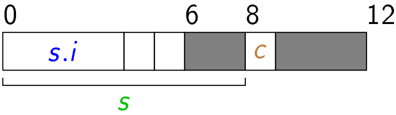
- Correct array cell alignment requires tail padding:



Representing heterogeneous data in memory

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

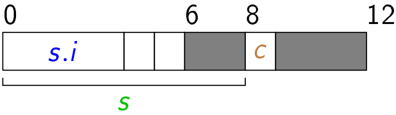
- A naive attempt:



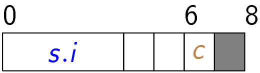
Representing heterogeneous data in memory

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

- A naive attempt:



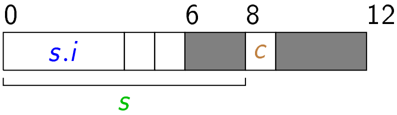
- Reuse tail padding in `s` to store `c`:



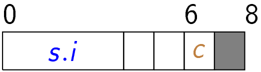
Representing heterogeneous data in memory

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

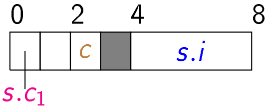
- A naive attempt:



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



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



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:

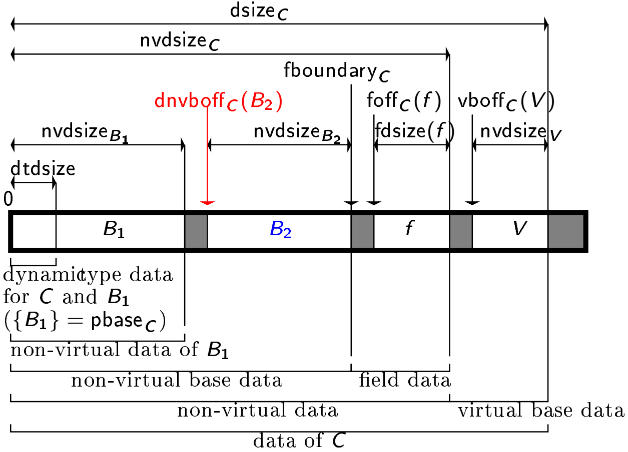
- alignment padding
- embedded structures: possibility of reusing padding?

Issues raised by multiple inheritance:

- Dynamic type data (e.g. pointers to virtual tables)
  - needed for dynamic cast, virtual function dispatch
  - accesses to virtual bases
  - 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.

Layout parameters

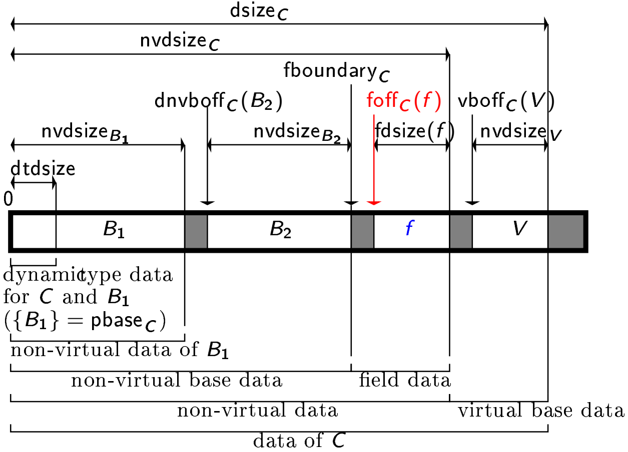
```
struct D: B1, B2,    virtual V { int f; };
```



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

Layout parameters

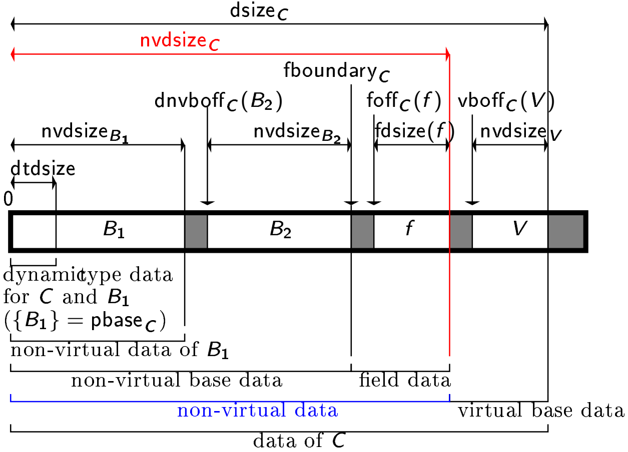
```
struct D: B1, B2,    virtual V { int f; };
```



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

Layout parameters

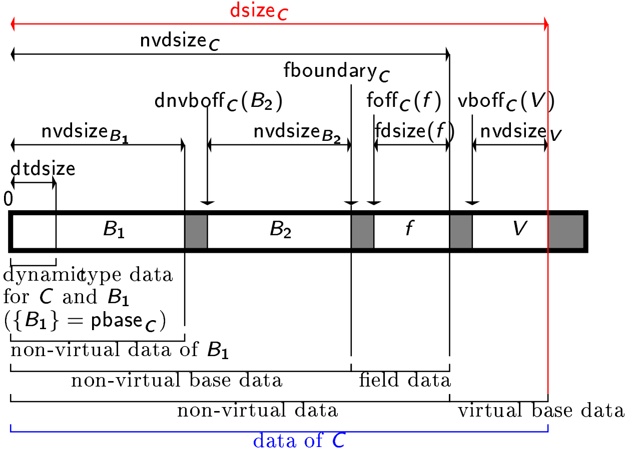
```
struct D: B1, B2,    virtual V { int f; };
```



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

Layout parameters

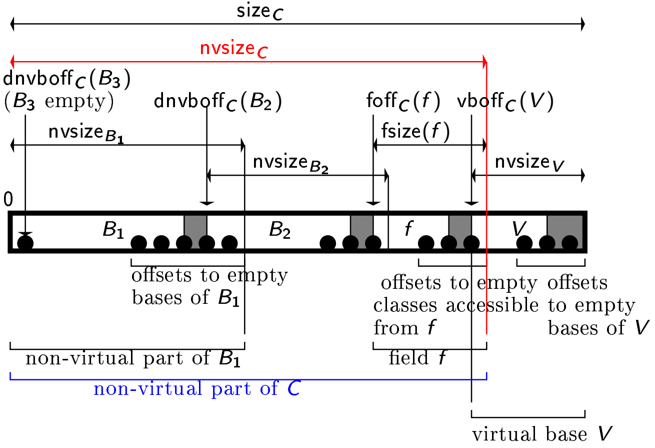
```
struct D: B1, B2,    virtual V { int f; };
```



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

Layout parameters

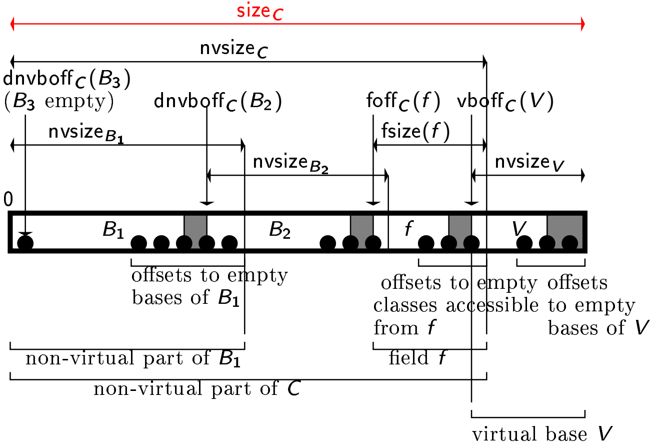
```
struct B3 { /* empty */ };
struct D: B1, B2, B3, virtual V { int f; };
```



nvsize_C is the total size of the **non-virtual** part of C

Layout parameters

```
struct B3 { /* empty */ };
struct D: B1, B2, B3, virtual V { int f; };
```



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: $$\begin{array}{cl} & [\mathrm{foff}_C(F_1), \mathrm{foff}_C(F_1) + \mathrm{fdatasize}(F_1)) \\ \# & [\mathrm{foff}_C(F_2), \mathrm{foff}_C(F_2) + \mathrm{fdatasize}(F_2)) \end{array}$$

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

Compilation of object-oriented operations

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

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

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

Compilation of object-oriented operations

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

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

+ static and dynamic casts, virtual function calls, ...

Set dynamic type: change the dynamic type data of an object and all its inheritance subobjects.

Outline

Semantics preservation

Theorem (Forward simulation)

Each transition step in the source program is simulated by one or several transition steps in the compiled program:

$$\begin{tikzcd} s_1 \arrow[r] \arrow[d, dash, "invariant"] & s_2 \arrow[d, dash, dotted, "invariant"] \\ s_1' \arrow[r, dotted, "+"] & s_2' \end{tikzcd}$$

Semantics preservation

Theorem (Forward simulation)

Each transition step in the source program is simulated by one or several transition steps in the compiled program:

$$\begin{tikzcd} s_1 \arrow[r] \arrow[d, dash, "invariant"] & s_2 \arrow[d, dash, dotted, "invariant"] \\ s'_1 \arrow[r, dotted, "+"] & s'_2 \end{tikzcd}$$

Corollary (Semantics preservation)

The compiler preserves the semantics of the source program: the compiled program has the same meaning as the source.

Outline

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:

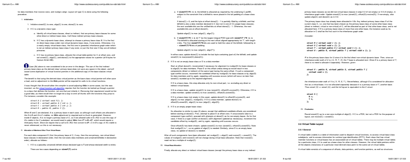
ptr:
: For a non-virtual function, this field is a simple function pointer. (Under current base Itanium psABI conventions, that is a pointer to a GP/function address pair.) For a virtual function, it is 1 plus the virtual table offset (in bytes) of the function, represented as a **ptrdiff_t**. The value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

The compiler can be used with this layout algorithm to obtain a verified compiler preserving the semantics of programs.

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

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

Outline

Assessment

- A general formal model for C++ object-oriented features
- First machine-checked formalization of RAII
- First machine-checked correctness proof of verified compiler for the C++ object model, including usual compiler techniques and realistic optimizations

Assessment

- A general formal model for C++ object-oriented features
- First machine-checked formalization of RAII
- First machine-checked correctness proof of verified compiler for the C++ object model, including usual compiler techniques and realistic optimizations
- Practical impact: positive feedback from C++ Standard Committee
  - standard issue corrected in C++11: virtual functions during destruction
  - other pending standard issues:
    - object lifetime and trivial constructors
    - lifetime of arrays
    - unification of destruction model for built-in types

Assessment

- A general formal model for C++ object-oriented features
- First machine-checked formalization of RAII
- First machine-checked correctness proof of verified compiler for the C++ object model, including usual compiler techniques and realistic optimizations
- Practical impact: positive feedback from C++ Standard Committee
  - standard issue corrected in C++11: virtual functions during destruction
  - other pending standard issues:
    - object lifetime and trivial constructors
    - lifetime of arrays
    - unification of destruction model for built-in types

Increased trust in C++ semantics and compilation.

Future work

Extending the semantics:

- Free store
- C++ copy semantics (passing constructor arguments by value, copy constructor, functions returning structures)
- Exceptions
- Templates

Improving the compiler:

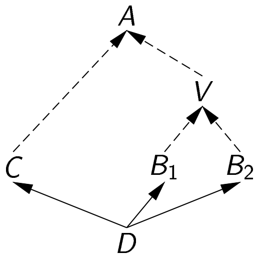
- Concrete representation of virtual tables and VTT
- Virtual primary bases
- Better object layout algorithms (bidirectional, etc.)

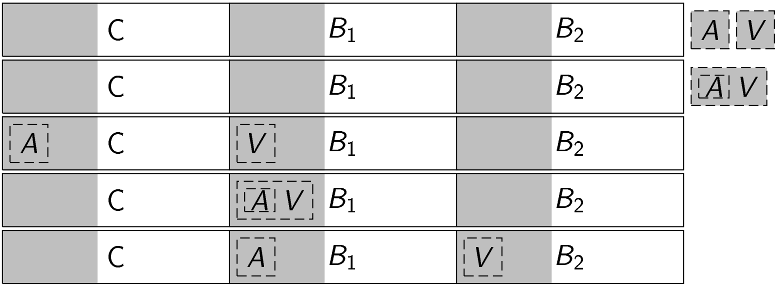
Thank you for your attention

- Coq development fully available on the Web: `http://gallium.inria.fr/~tramanan/cxx`
- For further information: `ramanana@nsup.org`

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





Destructing inherited objects

Java and C♯ are buggy:

```
class File implements Closeable {
  public void close () {...}
}
class BuggyFile extends File {
  public void close () {}
}

try (File f = new BuggyFile("toto.txt")) {
  ...
}
```

File is not closed properly. By contrast, C++ guarantees that destructors for base classes are called.

Sizes

(C1) $\mathrm{dnvboff}_C(B) + \mathrm{nvsize}_B \leq \mathrm{nvsize}_C$

if B direct non-virtual base of C

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

(C3) $\mathrm{vboff}_C(B) + \mathrm{nvsize}_B \leq \mathrm{size}_C$

if B generalized virtual base of C

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

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

Field separation

(C6) $[\mathrm{dnvboff}_C(B_1), \mathrm{dnvboff}_C(B_1) + \mathrm{nvdsize}_{B_1})$
$\# \; [\mathrm{dnvboff}_C(B_2), \mathrm{dnvboff}_C(B_2) + \mathrm{nvdsize}_{B_2})$
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) $[\mathrm{foff}_C(f_1), \mathrm{foff}_C(f_1) + \mathrm{fdsize}(f_1))$
$\# \; [\mathrm{foff}_C(f_2), \mathrm{foff}_C(f_2) + \mathrm{fdsize}(f_2))$
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) $[\mathrm{vboff}_C(B_1), \mathrm{vboff}_C(B_1) + \mathrm{nvdsize}_{B_1})$
$\# \; [\mathrm{vboff}_C(B_2), \mathrm{vboff}_C(B_2) + \mathrm{nvdsize}_{B_2})$
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

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

(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

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.

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`

Thank you for your attention

- ramanana@nsup.org
- http://gallium.inria.fr/~tramanan/cxx