Mechanized Formal Semantics and Verified Compilation for C++ objects

Tahina Ramananandro

\(^1\)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
Ariane 5 maiden flight (1996): US$370 million lost material and project delayed by 4 years due to overflow in floating-point computations.
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

```c
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

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

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

Source code

Proof that the program meets its specification
...and Verified Compilation...

Source code

Compilation

Machine-executable code

Proof that the program meets its specification
...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...
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

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
Initializing objects using a constructor

```c++
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

```cpp
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

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

```c
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

```c
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

```cpp
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

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
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 structure fields
  - static and dynamic casts
  - virtual function calls
  - object construction and destruction

- Properties of object construction and destruction

- A verified compiler to a minor-style 3-address language with low-level memory accesses
Outline

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

\[ n_{D,B} \ ::= \ D :: \cdots :: B \quad \text{Non-virtual inheritance path} \]

\[ p_{D,B} \ ::= \ (\text{Repeated}, n_{D,B}) \]
\[ \quad | \quad (\text{Shared}, n_{V,B}) \quad B \text{ is a non-virtual base of } D \]
\[ \quad | \quad (\text{Shared}, n_{V,B}) \quad V \text{ is a virtual base of } D \]
\[ \quad | \quad (\text{Shared}, n_{V,B}) \quad \text{and } B \text{ is a non-virtual base of } V \]
Designating subobjects with paths

We extended those works to embedded structures and arrays.

\[
\begin{align*}
\text{nv}_{D, B} & ::= D :: \cdots :: B & \text{Non-virtual inheritance path} \\
\rho_{D, B} & ::= (\text{Repeated}, \text{nv}_{D, B}) & B \text{ is a non-virtual base of } D \\
& | (\text{Shared}, \text{nv}_{V, B}) & V \text{ is a virtual base of } D \\
& & \text{and } B \text{ is a non-virtual base of } V \\
\text{subo} & ::= (idx, p, f) \cdots (idx', p') & \text{path to a subobject inside an array thru embedded structure array fields}
\end{align*}
\]
A core language

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

\[
Stmt \ ::= \ var \ ::= \ var \rightarrow_C f \\
| \quad \var \rightarrow_C f \ ::= \ var \\
| \quad \var \ ::= \ &\var[\var]_C \\
| \quad \var \ ::= \ static\_cast\langle A\rangle_C(\var) \\
| \quad \var \ ::= \ dynamic\_cast\langle A\rangle_C(\var) \\
| \quad \var \ ::= \ var \rightarrow_C f(\var, \ldots) \\
| \quad \{ C \var[n] = \{\ Init_C, \ldots \}; \ Stmt \} \\
| \quad \ldots \\
\]

\[
Init_C \ ::= \ Stmt; C(\var, \ldots)
\]

Reading scalar field or pointing to structure field
Writing scalar field
Pointing to array cell
Static cast
Dynamic cast
Virtual function call
Block-scoped object
Structured control
Initializer
A core language

\[
\begin{align*}
\text{Funct} & \ ::= \ \text{virtual} \ f(\text{var}, \ldots)\{\text{Stmt}\} \\
\text{Finit}_m & \ ::= \\
& \quad m\{\text{Init}_A \ldots\} \\
& \quad | \quad m(\text{Stmt}, \text{var}) \\
\text{Constr}_C & \ ::= \ C(\text{var}, \ldots) : \text{Init}_{B_1}, \ldots, \text{Init}_{V_1}, \ldots, \\
\text{Destr}_C & \ ::= \ \sim C()\{\text{Stmt}\} \\
\text{Finit}_m, \ldots \{\text{Stmt}\} \\
\text{Class} & \ ::= \ \text{struct} \ C : B_1, \ldots, \text{virtual} \ V_1, \ldots \\
& \quad \{\text{Constr}_C \ldots; \text{Funct} \ldots ; \text{Destr}_C\} \\
\text{Prog} & \ ::= \ \text{Class} \ldots
\end{align*}
\]

Virtual function
Data member initializers
Structure
Scalar
Constructor
Destructor
Class definition
Program
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 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.
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:

- Unconstructed
- StartedConstructing
- BasesConstructed
- Constructed
- StartedDestructing
- DestructingBases
- Destructed
Evolution of the construction state during construction

```
struct C : B {
    int i;

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

Unconstructed
```
Evolution of the construction state during construction

```cpp
struct C : B {
    int i;

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

StartedConstructing
```
Evolution of the construction state during construction

```cpp
struct C : B {
    int i;

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

BasesConstructed, virtual functions allowed here
Evolution of the construction state during construction

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

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:

<table>
<thead>
<tr>
<th>If ( p' ) is...</th>
<th>Then ( p ) is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconstructed</td>
<td>Unconstructed</td>
</tr>
<tr>
<td>StartedConstructing</td>
<td>Unconstructed</td>
</tr>
</tbody>
</table>
|                   | \( \text{if } p \text{ is a field subobject of } p' \)  
|                   | between Unconstructed and Constructed  
|                   | otherwise         |
| Bases Constructed | Constructed       |
|                   | \( \text{if } p \text{ is a base subobject of } p' \)  
|                   | between Unconstructed and Constructed  
|                   | otherwise         |
| Constructed       | Constructed       |
| Started Destructing | Constructed       |
|                   | \( \text{if } p \text{ is a base subobject of } p' \)  
|                   | between Constructed and Destructed  
|                   | otherwise         |
| Destructing Bases | Destructed        |
|                   | \( \text{if } p \text{ is a field subobject of } p' \)  
|                   | between Constructed and Destructed  
|                   | otherwise         |
| Destructed        | Destructed        |

\[ \text{Diagram: } V \begin{array}{c} D \end{array} \begin{array}{c} D \end{array} \begin{array}{c} V \end{array} \begin{array}{c} X \end{array} \begin{array}{c} B_1 \end{array} \begin{array}{c} B_2 \end{array} \begin{array}{c} p \end{array} \begin{array}{c} A \end{array} \begin{array}{c} A \end{array} \begin{array}{c} p' \end{array} \]
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:

<table>
<thead>
<tr>
<th>If ( p_1 ) is...</th>
<th>Then ( p_2 ) is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconstructed</td>
<td>Unconstructed</td>
</tr>
<tr>
<td>StartedConstructing</td>
<td>Unconstructed</td>
</tr>
<tr>
<td>BasesConstructed</td>
<td></td>
</tr>
<tr>
<td>Constructed</td>
<td>in an arbitrary state</td>
</tr>
<tr>
<td>StartedDestructing</td>
<td></td>
</tr>
<tr>
<td>DestructingBases</td>
<td>Destructed</td>
</tr>
<tr>
<td>Destructed</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image)
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

```c++
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

```cpp
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

\[ \text{struct C : B \{ ... \};} \]

- 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

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

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

Destructors
Outline

1. Overview of the C++ object model

2. Formal semantics

3. Verified compilation
   - Compilation of constructors and destructors
   - Formalization of C++ object layout
   - Semantics preservation
   - Real-world layout algorithms

4. Conclusion and perspectives
Compilation of object constructors and destructors

Constructors → Set dynamic type → Low-level memory accesses

Destructors
Compilation of object constructors and destructors

```c
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;
}
```
Compilation of object constructors and destructors

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

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

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

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 = D();
    ...
    return 42;
}
```

Ramananandro (INRIA)
Compilation of object constructors and destructors

```c++
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;
}
```

```c++
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");
}
```

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

1. Overview of the C++ object model
2. Formal semantics
3. Verified compilation
   - Compilation of constructors and destructors
   - Formalization of C++ object layout
   - Semantics preservation
   - Real-world layout algorithms
4. Conclusion and perspectives
Representing C++ objects in concrete memory

Constructors → Set dynamic type → Low-level memory accesses

Destructors
Representing homogeneous data in memory

\[
\begin{bmatrix}
11 & 22 & 33 \\
\end{bmatrix}
\]

0 \hspace{1cm} 4 \hspace{1cm} 8 \hspace{1cm} 12

\begin{array}
0 & 11 & 22 & 33
\end{array}
Representing homogeneous data in memory

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

\[
\begin{array}{c c c}
0 & 4 & 8 \\
11 & 22 & 33
\end{array}
\]

\[
\begin{array}{c c c c c}
0 & 12 & 24 \\
11 & 22 & 33 & 44 & 55 & 66
\end{array}
\]
Representing homogeneous data in memory

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

\[
\begin{bmatrix}
11 & 22 & 33 & 44 & 55 & 66 \\
11 & 22 & 33 & 44 & 55 & 66 \\
11 & 44 & 22 & 55 & 33 & 66 \\
\end{bmatrix}
\]
Representing heterogeneous data in memory

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

- A naive representation:

```
 0 1  5 6
```

```
c1  i  c2
```

Correct field alignment requires padding:

```
 0 1  5 6
```

Reorder fields to save space:

```
 0 4  5 6
```

A naive attempt:

```
s.i  0 6 8
```

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

```
s.i  0 6 8
```

Reorder fields in `s` to reuse inside padding:

```
 0 2  4 8
```

```
s.c1 1 4 5 6
```
Representing heterogeneous data in memory

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

- A naive representation:

```
  0  1  x
  c1  i  c2
```

Correct field alignment requires padding:

```
  0  1  5  6
  c1  i  c2
```

Rearrange fields to save space:

```
  i  c1  c2
```

A naive attempt:

```
  s.i  6  8
  s.c1 12
```

Reuse tail padding in `s` to store `c2`:

```
  s.i  6  8
  s.c1 12
```

Rearrange fields in `s` to reuse inside padding:

```
  c1  s.i  8
  s.c1 24
```
Representing heterogeneous data in memory

```
struct S { char c1; int i; char c2; };
```

- A naive representation:

```
0 1 X 5 6
C1 i C2
```

- Correct field alignment requires padding:

```
0 1 4 ✓ 8 9
C1 i C2
```
Representing heterogeneous data in memory

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

- A naive representation:
  
  ![Naive representation diagram]

- Correct field alignment requires padding:
  
  ![Correct alignment diagram]

- Reorder fields to save space:
  
  ![Reordered fields diagram]
Representing heterogeneous data in memory

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

Making arrays of structures:

- Correct array cell alignment requires tail padding:

```c
s[0].i  |  s[1].i  |  6  |  12
```

- A naive attempt:

```c
s[0].c  |  0  |  6  |  12
```

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

```c
s[0].c  |  2  |  4  |  8
```
Representing heterogeneous data in memory

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

- Making arrays of structures:

```
0   6   12
s[0].i  s[1].i
```

- Correct array cell alignment requires tail padding:

```
0   6   12
s[0].i  s[1].i
```

- A naive attempt:

```
0   6   12
s[0].i  s[1].i
```

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

```
0   6   12
s[0].i  s[1].i
```

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

```
0   2   8
s[0].c1  s[1].c
```

```
0   2   4   8
s[0].c1  s[1].c
```
Representing heterogeneous data in memory

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

- Making arrays of structures:

```
0   6  x
s[0].i  s[1].i
s[0]
```

- Correct array cell alignment requires tail padding:

```
0   6   8  x
s[0].i  (pad)  s[1].i  (pad)
s[0]
```
Representing heterogeneous data in memory

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

- A naive attempt:

```plaintext
A naive attempt:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>6</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>s.i</td>
</tr>
<tr>
<td></td>
<td>s</td>
<td></td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>
```

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

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

```plaintext
Ramananandro (INRIA)
Les objets en C++: . . .
January 10th, 2012 42 / 54```
Representing heterogeneous data in memory

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

- A naive attempt:

```
0 6 8 12
```
```
<table>
<thead>
<tr>
<th></th>
<th>s.i</th>
<th></th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Reuse tail padding in s to store c:

```
0 6 8
```
```
<table>
<thead>
<tr>
<th></th>
<th>s.i</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Representing heterogeneous data in memory

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

- A naive attempt:

  ![Naive Attempt Diagram]

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

  ![Reuse Tail Padding Diagram]

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

  ![Reorder Fields Diagram]
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.
La y out pa rameters

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

dsize_C

nvdsze_C

fboundary_C

dnvboff_C(B_2)

foff_C(f)

vboff_C(V)

fv

C(f)

dsize(f)

V

B1

B2

f

V

dynamic type data for C and B1

non-virtual data of B1

non-virtual base data

field data

non-virtual data

virtual base data

data of C

dnvboff_C(B) is the offset, within C, of the direct non-virtual base B of C
struct D: B1, B2, virtual V { int f; };
struct D: B1, B2, virtual V { int f; };

nvdatasize\textsubscript{C} is the data size of the **non-virtual** part of \( C \), excluding its tail padding
struct D: B1, B2, virtual V { int f; };

C is the data size of a full C object, excluding its tail padding.
Layout parameters

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

![Diagram of class layout]

\( \text{nvsize}_C \) is the total size of the **non-virtual** part of \( C \)
La y out pa rameters

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**: $\text{foff}_C(F) + \text{fsize}(F) \leq \text{nsize}_C$

- **C9**: 
  $\begin{align*}
  \text{foff}_C(F_1), \text{foff}_C(F_1) + \text{fdatasize}(F_1) \\
  \# \quad \text{foff}_C(F_2), \text{foff}_C(F_2) + \text{fdatasize}(F_2)
  \end{align*}$

Those conditions do not deterministically fix the offsets and sizes, it is up to the algorithm.
Compilation of object-oriented operations

\[
[x := x' \rightarrow_{C} F] = x := \text{load}(\text{scsize}_t, x' + \text{foff}_C(F))
\]

(if \( F = (f, t) \) is a scalar field of \( C \))

\[
[x \rightarrow_{C} F := x'] = \text{store}(\text{scsize}_t, x + \text{foff}_C(F), x')
\]

(if \( F = (f, t) \) is a scalar field of \( C \))

\[
[x := x' \rightarrow_{C} F] = x := x' + \text{foff}_C(F)
\]

(if \( F \) is a structure array field of \( C \))
Compilation of object-oriented operations

\[ x := x_1[x_2]_C \] = \( x := x_1 + \text{size}_C \times x_2 \)

\[ x := x_1 == x_2 \] = \( x := x_1 == x_2 \)

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

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

1. Overview of the C++ object model
2. Formal semantics
3. Verified compilation
   - Compilation of constructors and destructors
   - Formalization of C++ object layout
   - Semantics preservation
   - Real-world layout algorithms
4. Conclusion and perspectives
Semantics preservation

**Theorem (Forward simulation)**

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

\[
S_1 \xrightarrow{\text{invariant}} S_2 \\
S'_1 \xrightarrow{+} S'_2
\]
Theorem (Forward simulation)

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

$$
\begin{align*}
S_1 \rightarrow & \rightarrow S_2 \\
| \quad \text{invariant} \quad | \text{invariant} \\
S_1' \rightarrow & \rightarrow S_2'
\end{align*}
$$

Corollary (Semantics preservation)

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

1. Overview of the C++ object model

2. Formal semantics

3. Verified compilation
   - Compilation of constructors and destructors
   - Formalization of C++ object layout
   - Semantics preservation
   - 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
- 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

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.
- nsize(O): the non-virtual size of an object, which is the size of O without virtual bases.
- nalign(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

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

Increased trust in C++ semantics and compilation.

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
  ⋆ communication 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
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

```cpp
struct A { virtual void f(); }
struct V : virtual A
struct C : virtual A
struct B1 : virtual V
struct B2 : virtual V
struct D : C, B1, B2
```
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) \( \text{dnvboff}_C(B) + \text{nvsize}_B \leq \text{nvsize}_C \) 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) \[ [\text{dnvboff}_C(B_1), \text{dnvboff}_C(B_1) + \text{nvdsiz}_B] \]
\[ \neq [\text{dnvboff}_C(B_2), \text{dnvboff}_C(B_2) + \text{nvdsiz}_B] \]
if \( B_1, B_2 \) distinct non-empty non-virtual direct bases of \( C \)

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

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

(C9) \[ [\text{foff}_C(f_1), \text{foff}_C(f_1) + \text{fdsize}(f_1)] \]
\[ \neq [\text{foff}_C(f_2), \text{foff}_C(f_2) + \text{fdsize}(f_2)] \]
if \( f_1 \) and \( f_2 \) are distinct relevant fields of \( C \)

(C10) \( \text{foff}_C(f) + \text{fdsize}(f) \leq \text{nvdsiz}_C \)
if \( f \) relevant field of \( C \)

(C11) \( \text{fboundary}_C \leq \text{nvdsiz}_C \)

(C12) \[ [\text{vboff}_C(B_1), \text{vboff}_C(B_1) + \text{nvdsiz}_B] \]
\[ \neq [\text{vboff}_C(B_2), \text{vboff}_C(B_2) + \text{nvdsiz}_B] \]
if \( B_1, B_2 \) distinct non-empty generalized virtual bases of \( C \)

(C13) \( \text{vboff}_C(B) + \text{nvdsiz}_B \leq \text{dsiz}_C \)
Field alignment – Dynamic type data

Field alignment

(C14) \((\text{falign}(f) \mid \text{foff}_C(f)) \text{ and } (\text{falign}(f) \mid \text{nalign}_C)\) 
if \(f\) field of \(C\)

(C15) \((\text{nalign}_B \mid \text{dnvboff}_C(B)) \text{ and } (\text{nalign}_B \mid \text{nalign}_C)\) 
if \(B\) non-virtual base of \(C\)

(C16) \((\text{dtdalign} \mid \text{nalign}_C)\) 
if \(C\) is dynamic

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

(C18) \((\text{align}_C \mid \text{size}_C)\)

Dynamic type data

(C19) \(\text{dtdsize} \leq \text{fboundary}_C\)

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

(C21) \(\text{pbase}_C = \{B\} \Rightarrow \text{dnvboff}_C(B) = 0\)
Identity of subobjects

\[
\text{eboffs}_C = \begin{cases} \text{def} & \bigcup \text{vboff}_C(B) + \text{nveboffs}_B \\
B \in \text{vbases}_C \cup \{C\} \end{cases}
\]

\[
\text{nveboffs}_C = \begin{cases} \text{def} & \text{if } C \text{ is empty then } \{(C, 0)\} \text{ else } \emptyset \\
\bigcup \bigcup \text{dnevboffs}_C(B) + \text{nveboffs}_B \\
B \in \text{dnevbases}_C \end{cases}
\]

\[
\bigcup \bigcup \text{foff}_C(f, B, n) + i \cdot \text{size}_B + \text{eboffs}_B \\
(f, B, n) \in \text{stfields}_C \\
0 \leq i < n
\]

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

(C23) (dnevboff\(_C(B_1) + \text{nveboffs}_B_1) \# (\text{dnevboff}_C(B_2) + \text{nveboffs}_B_2)

if \( B_1, B_2 \) distinct non-virtual bases of \( C \)

(C24) (dnevboff\(_C(B_1) + \text{nveboffs}_B_1) \# \bigcup \text{foff}_C(f, B_2, n) + j \cdot \text{size}_B + \text{eboffs}_B_2 \\
0 \leq j < n

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

(C25) \bigcup \text{foff}_C(f_1, B_1, n_1) + j_1 \cdot \text{size}_B + \text{eboffs}_B_1 \\
0 \leq j_1 < n_1

\# \bigcup \text{foff}_C(f_2, B_2, n_2) + j_2 \cdot \text{size}_B + \text{eboffs}_B_2 \\
0 \leq j_2 < n_2

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

(C26) (vboff\(_C(B_1) + \text{nveboffs}_B_1) \# (vboff\(_C(B_2) + \text{nveboffs}_B_2)

if \( B_1, B_2 \) distinct generalized virtual bases of \( C \)
Excerpt from Mark Mitchell,

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 aggressive, fail to enforce object identity (http://www.cantrip.org/emptyopt.html)


```c
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

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