Compiler Construction: Practical Introduction
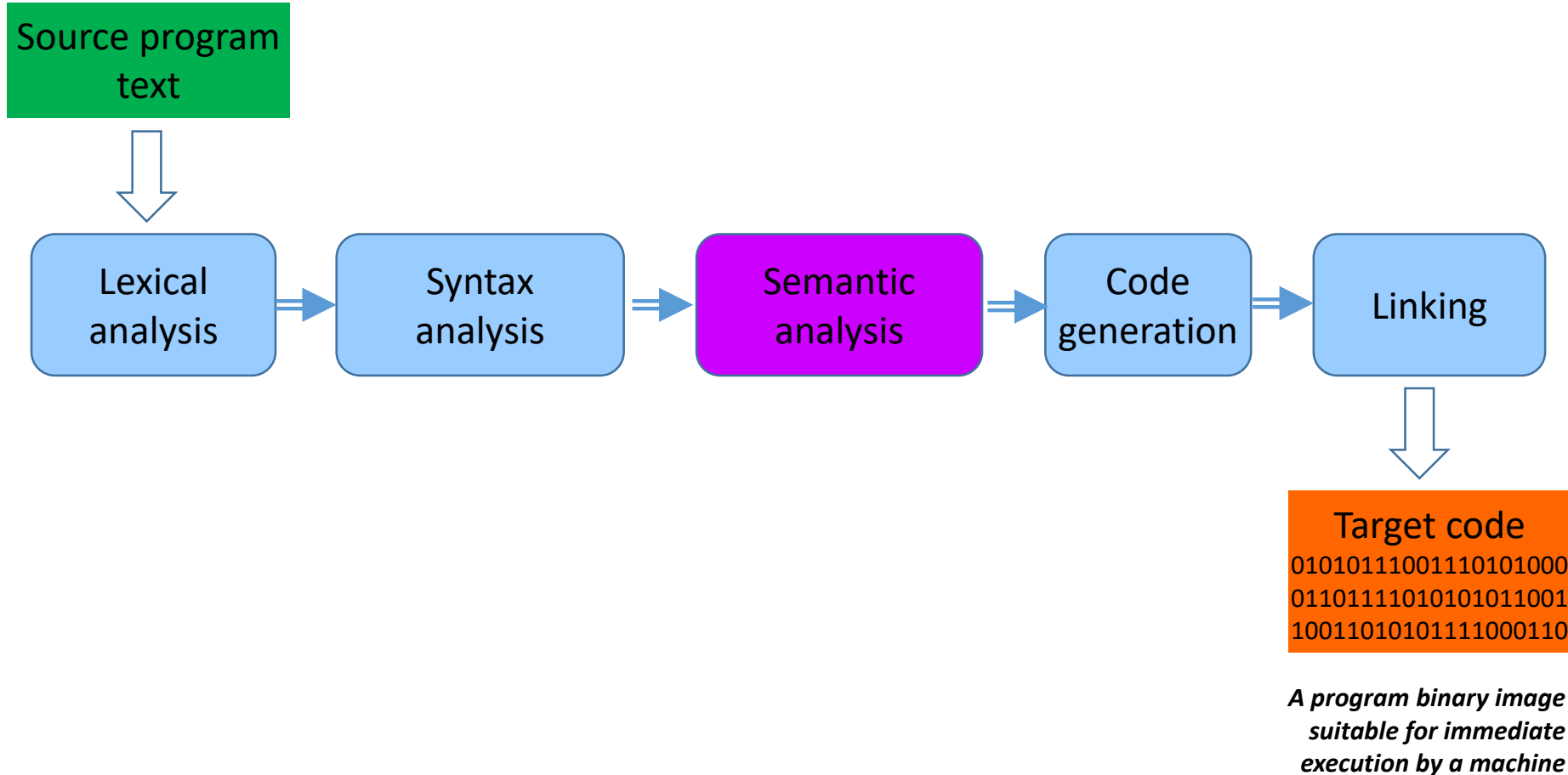
Lecture 7
Semantic Analysis

Eugene Zouev

Spring Semester 2022
Innopolis University

Main Topics

- Why and for semantic analysis is?

- Examples: standard conversions, initialization semantics, user-defined conversions, calculating constant expressions

- Source code optimizations: the general idea

- Examples: eliminating repeated calculations, replacing slow instructions, excluding redundant calculations, constant propagation etc.
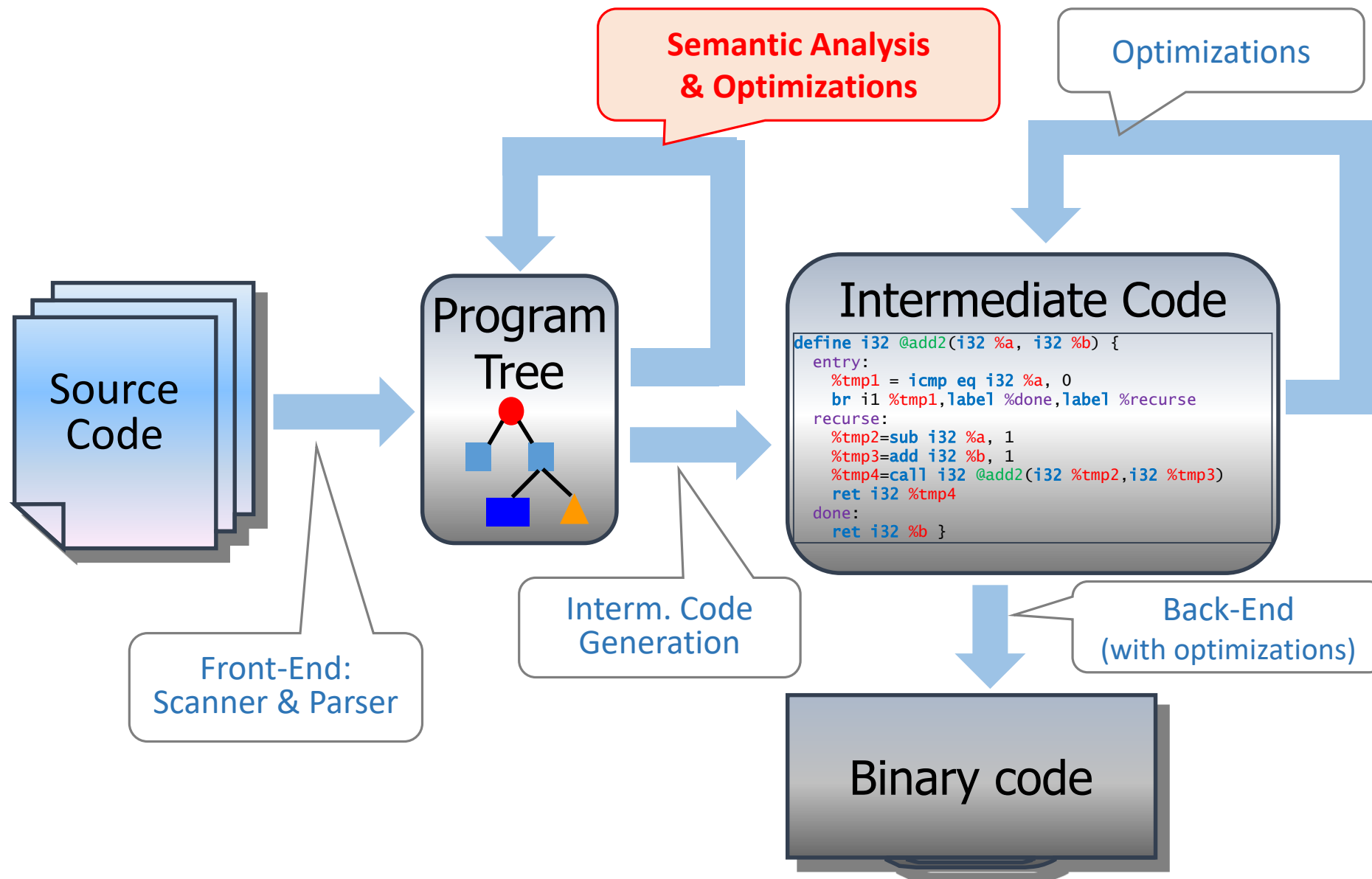
Compilation: An Ideal Picture

*A program written by a human
(or by another program)*

Source program text

↓

Lexical analysis → Syntax analysis → Semantic analysis → Code generation → Linking

↓

Target code
0101011100111010100 0
011011110101010110 01
1001101010111100011 0

*A program binary image
suitable for immediate
execution by a machine*

Where We Are Today

Coming back to the today's topic

Semantic Analysis & Optimizations

Optimizations

Source Code

Program Tree

Intermediate Code

```
define i32 @add2(i32 %a, i32 %b) {
  entry:
    %tmp1 = icmp eq i32 %a, 0
    br i1 %tmp1,label %done,label %recurse
  recurse:
    %tmp2=sub i32 %a, 1
    %tmp3=add i32 %b, 1
    %tmp4=call i32 @add2(i32 %tmp2,i32 %tmp3)
    ret i32 %tmp4
  done:
    ret i32 %b }
```

Front-End: Scanner & Parser

Interm. Code Generation

Back-End (with optimizations)

Binary code

What Is Semantic Analysis For?

```
void f(int p)
{
    int a, b;
    *a = 777;
    return xyz*a+b*f;
}
...
f();
int x = f(1,2);
```

What Is Semantic Analysis For?

```
void f(int p)
{
    int a, b;
    *a = 777;
    return xyz*a+b*f;
}
...
f();
int x = f(1,2);
```

Syntactically
perfect program!..

What Is Semantic Analysis For?

```
void f(int p)
{
    int a, b;
    *a = 777;
    return xyz*a+b*f;
}
...
f();
int x = f(1,2);
```

Syntactically perfect program!..

Illegal operation (dereferencing) for the object of type **int**

Using uninitialized variables a and b

Undeclared variable xyz

Illegal operand types for operator *

Returning a value in **void** function

Illegal number of arguments in calls to function f

Illegal position for call to function f

What Is Semantic Analysis For?

Some remarks

1. Errors like "undeclared identifier" are typically detected on syntax analysis stage – while building symbol tables and/or program tree.

2. Errors like "uninitialized variable" usually are not detected by all compilers because it requires deeper control flow and data flow analysis.

3. Analysis of the code snippet …xyz*a… typically results in a message like "illegal operand types for * operator". Formally that's true but in fact the reason is that xyz is not declared – this is an example of and *induced error*.

Наведенные ошибки

Semantic Analysis

- Typically semantic analysis runs <u>on the program tree</u> built on previous compilation stages (while syntax analysis).

- Semantic analysis is typically implemented as <u>a series of tree traverses</u> with some actions related to the source language semantics.

- The more complex semantics is <u>the more passes</u> (traverses) are needed.

  - For relatively simple languages semantic analysis can be done **together** with syntax analysis while building the program tree.

  - Usually, <u>the last tree walk implements target code generation</u> – either an intermediate representation (like C--) or assembler code.

  - Often, before code generation, some **additional stages** after semantic analysis are necessary like building CFG & SSA representations…

Semantic Analysis

- One or several semantic actions are performed on each tree walk.

- What's the result of each tree walk?

  - Either a modified program tree with **the same node types**; perhaps <u>complex nodes get replaced for simpler ones</u>.

    Example is the C# compiler: after each tree walk the tree consists of the same node types.

  - Or a modified program tree **with different node types** that are <u>more primitive</u> but are "closer" to the target architecture.

    Example is the Scala compiler: node types representing source program constructs get replaced for more primitive nodes ("ICode"), and the JVM (or MSIL) code is generated from ICode finally.

Semantic Analysis

The result of each tree walk is typically twofold:

- The tree <u>changes its structure</u>: some nodes/subtrees are added or removed, some nodes/subtrees get replaced for other nodes/subtrees…

- Tree nodes <u>are annotated</u> ("decorated" ☺) <u>by attributes</u> reflecting various semantic features; the attributes are deduced during the analysis process.

=> The Abstract Syntax Tree (AST) is converted
   to the **Annotated** Syntax Tree (AAST).

   (An alternative solution is **attribute grammars.**)

Semantic Analysis: Actions

Four categories of actions while semantic analysis:

- Semantic checks

  Operand types consistency in expressions
  Checking correctness for function calls (including destructors)

- Semantic conversions

  Replacing conversions for function calls

  Replacing infix operators for operator function calls

  Inserting necessary type conversions

  Template instantiating

- Identification of hidden semantics

  Implicit destructor calls
  Temporary objects

- Optimizations (!)

AAST example (a fragment)

```
class C { ... };
float f(const char* s)
{
    C c(s); return c.m;
}
```

CLASS_DECL
"C"
CLASS_BODY → ...

FUNC_DECL
"f"
FLOAT
PARAMS

FUNC_BODY
VAR_DECL → ID
"c" CLASS

RET_STMT → MEMB_SELEC → ID
ID

PARM_DECL
"s" PTR → CONST → CHAR

- **Structural links**
- **Type information**
- **Attributes**

Ext1

AAST example (a fragment)

From the prev lecture

class C { ... };
float f(const char* s)
{
    C c(s); return c.m;
}

CLASS_DECL

"C"   CLASS_BODY → ...

FUNC_DECL

"f"

FLOAT

PARAMS

FUNC_BODY

VAR_DECL → ID

"c"   CLASS

RET_STMT → MEMB_SELEC → ID

ID

PARM_DECL
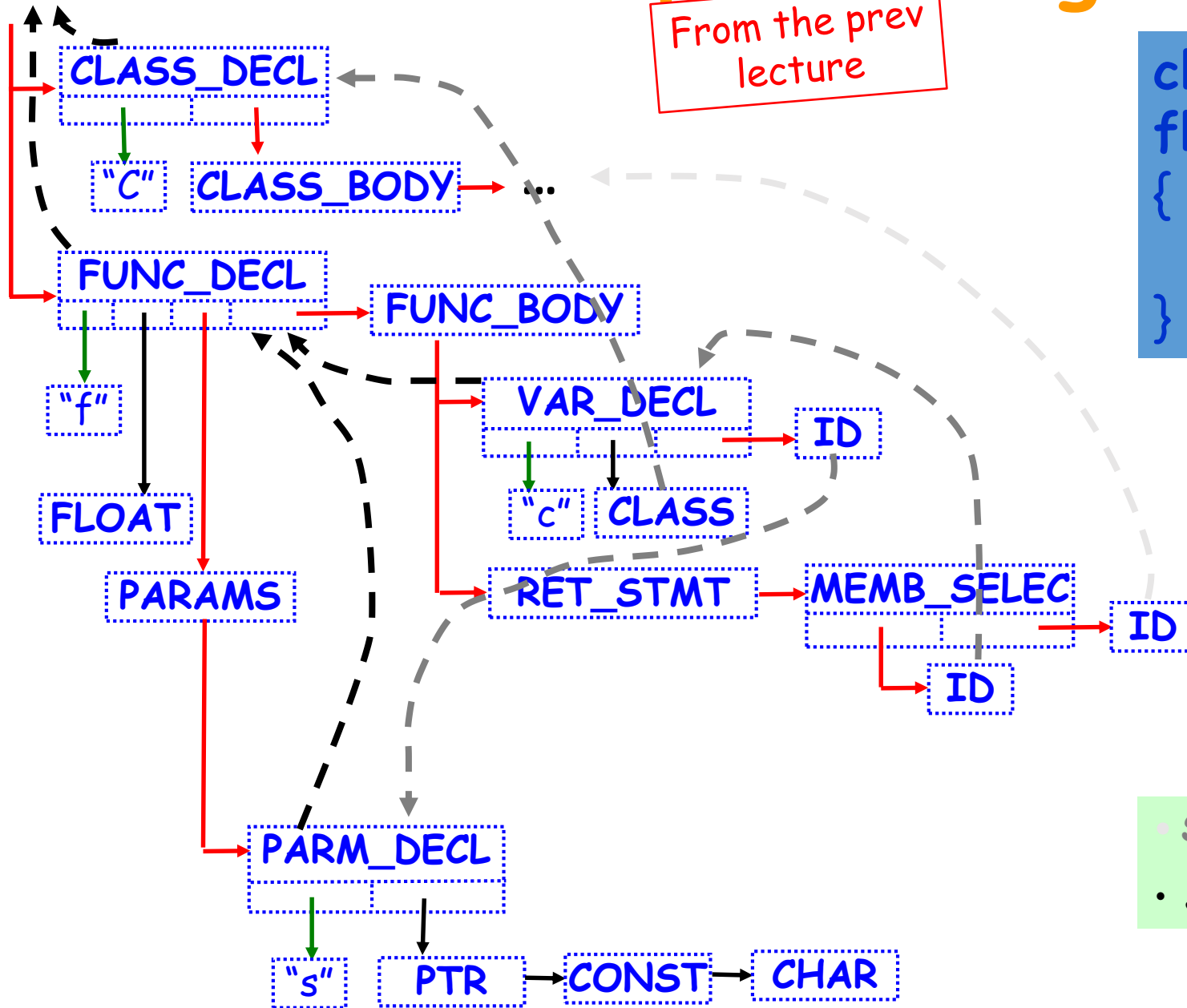
"s"   PTR → CONST → CHAR

- Semantic links

- **Scopes**

Ext2

AAST example (a fragment)

CLASS_DECL

"C" CLASS_BODY → ...

FUNC_DECL → FUNC_BODY

"f"

FLOAT

PARAMS

VAR_DECL → ID

"c" CLASS

RET_STMT → MEMB_SELEC → ID

ID

DTOR_CALL

ID

PARM_DECL
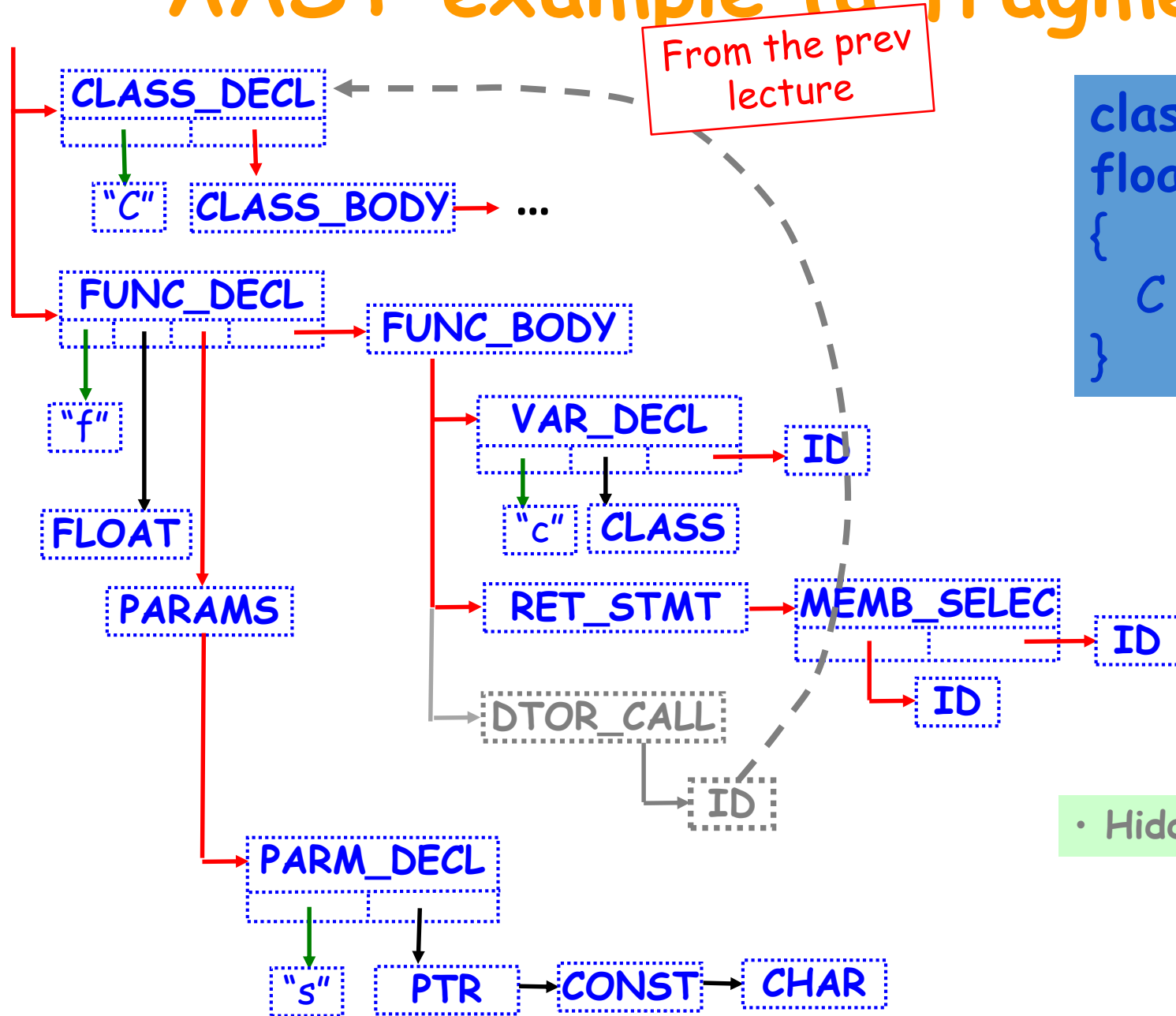
"s" PTR → CONST → CHAR

```
class C { ... };
float f(const char* s)
{
    C c(s); return c.m;
}
```

- Hidden semantics

Ext3

Standard conversions

```
int a = 3;
double d = 7.55;
float x = a + d;
```

Standard conversions

```
int a = 3;
double d = 7.55;
float x = a + d;
```

```
PLUS
```

```
REF   a
```

```
REF   d
```

```
...
float x = (float)((double)a + d);
```

Semantic Analysis: Example 1

Standard conversions

int a = 3;
double d = 7.55;
float x = a + d;

...

float x = (float)((double)a + d);

PLUS

REF a

REF d

CONV
tpFloat

PLUS
tpDouble

CONV
tpDouble

REF d
tpDouble

REF a
tpInt

Class declaration

```
class C {
    public:
        int a, b;
};
```

Class declaration

```
class C {
    public:
        int a, b;
};
```



```
class C {
    public:
        C() { a = 0; b = 0; }
        C(const C& c) { a = c.a; b = c.b; }
        ...
    int a, b;
};
```

Automatically generated:
Default constructor
Copy constructor
Move constructor
Copy assignment operator
...

CLASS C   MEMB a   MEMB d
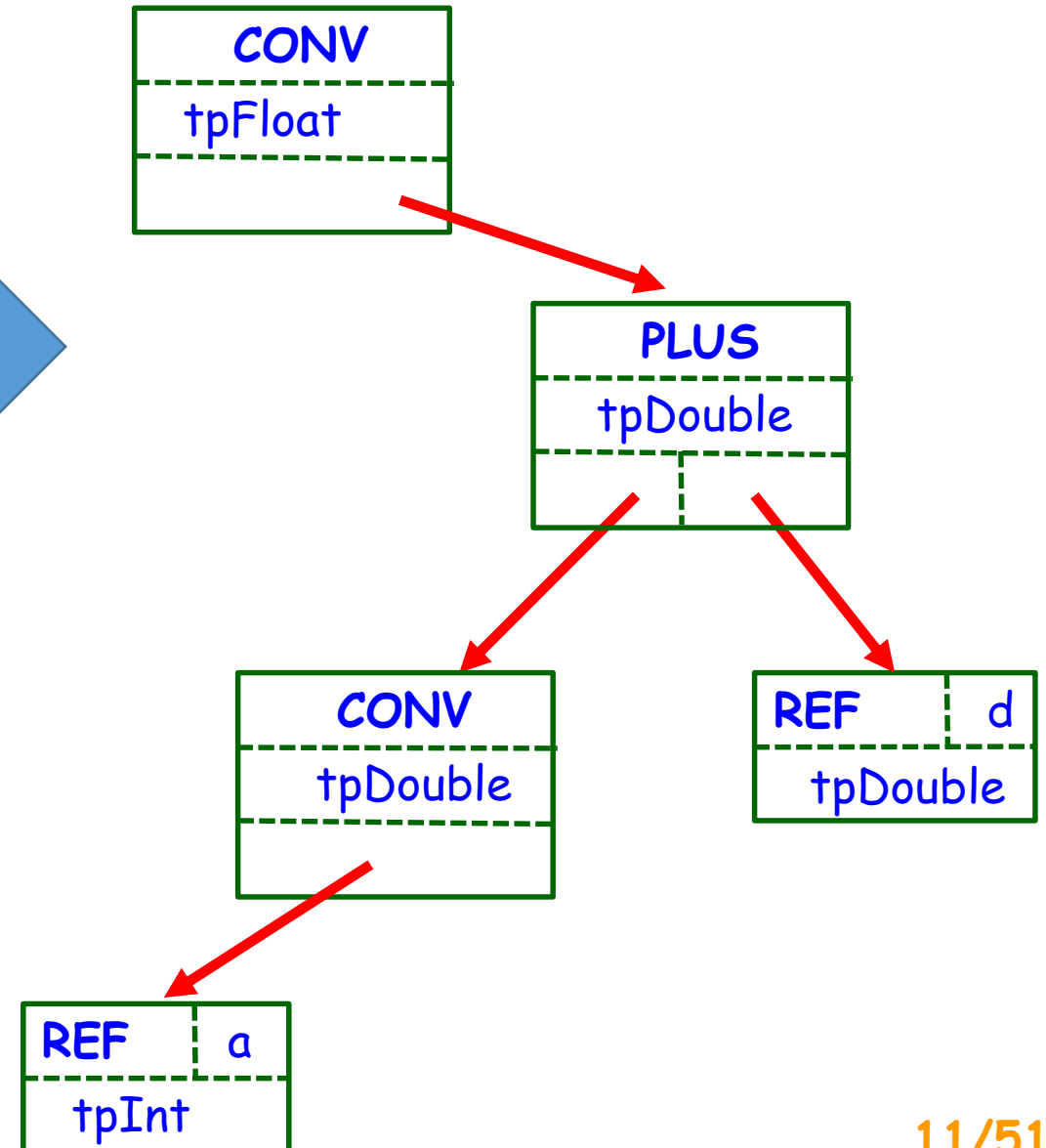
Semantic Analysis: Example 2

Class declaration

```
class C {
    public:
        int a, b;
};
```

```
class C {
    public:
        C() { a = 0; b = 0; }
        C(const C& c) { a = c.a; b = c.b; }
        ...
    int a, b;
};
```

Automatically generated:
   Default constructor
   Copy constructor
   Move constructor
   Copy assignment operator
   ...

| CLASS C | MEMB a | MEMB d |

| CLASS C | CTOR | CTOR | ... |

Parameters   Body

Parameters   Body

| ... | MEMB a | MEMB d |
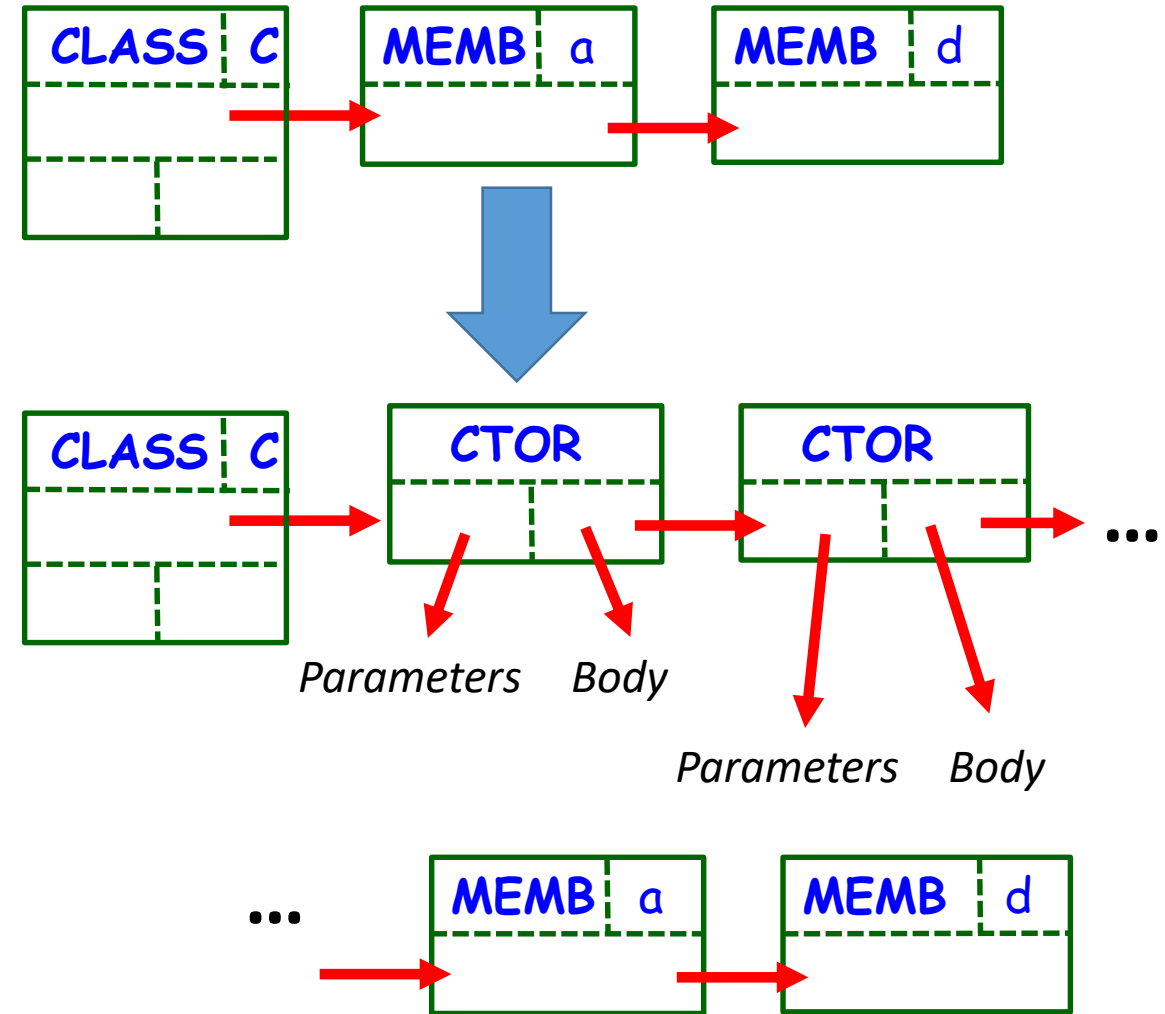
Initialization semantics

```
class C { ... };
...
C c1;
C c2(1);
C c3(c2);
C c4 = 7;
C c5 = c1;
```

Semantic Analysis: Example 3

Initialization semantics

- Allocate memory for c1 object;
- Call **default constructor** of C for c1.

class C { ... };
...
C c1;
C c2(1);
C c3(c2);
C c4 = 7;
C c5 = c1;

Initialization semantics

- Allocate memory for c1 object;
- Call **default constructor** of C for c1.

```
class C { ... };
...
C c1;
C c2(1);
C c3(c2);
C c4 = 7;
C c5 = c1;
```

- Allocate memory for c2 object;
- Call **C(int)** constructor for c2.

Semantic Analysis: Example 3

Initialization semantics

```
class C { ... };
...
C c1;
C c2(1);
C c3(c2);
C c4 = 7;
C c5 = c1;
```
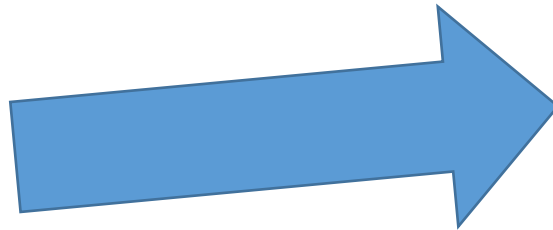
- Allocate memory for c1 object;
- Call **default constructor** of C for c1.

- Allocate memory for c2 object;
- Call **C(int)** constructor for c2.

- Allocate memory for c3 object;
- Call **copy constructor** for c3.
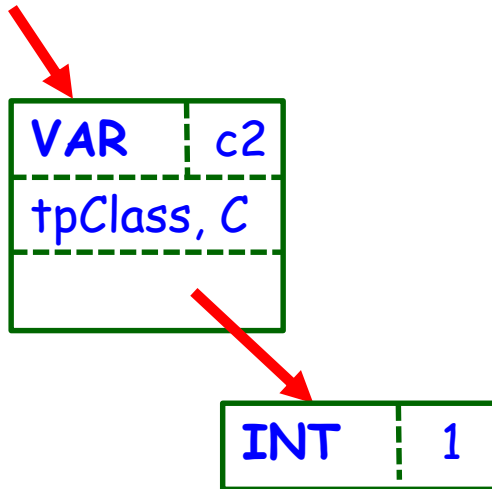
Initialization semantics

```
class C { ... };
...
C c2(1);
```

...

```
+----------+----+
| VAR      | c2 |
+----------+----+
| tpClass, C    |
+----------+----+
|               |
+---------------+

          +------+----+
          | INT  | 1  |
          +------+----+
```

Initialization semantics

class C { ... };
...
C c2(1);

...

VAR	c2
tpClass, C	

CALL	

VAR	c2
tpClass, C	

INT	1

REF	C(int)

ADDR	
tpPtr, tpClass, C	

INT	1
tpInt	

REF	c2
tpClass, C	

User-defined conversions

```
class C {
private:
    bool m;
public:
    operator bool() { return m; }
};
...
C c;
...
if ( c ) ...
```

User-defined conversions

```
class C {
private:
   bool m;
public:
   operator bool() { return m; }
};
...
C c;
...
if ( c ) ...        ⟵  if ( (bool)c ) ...
```

User-defined conversions

```
class C {
private:
   bool m;
public:
   operator bool() { return m; }
};
...
C c;
...
if ( c ) ...  ◄── if ( (bool)c ) ...  ◄── if ( c.operator bool() ) ...
```

if (c) …
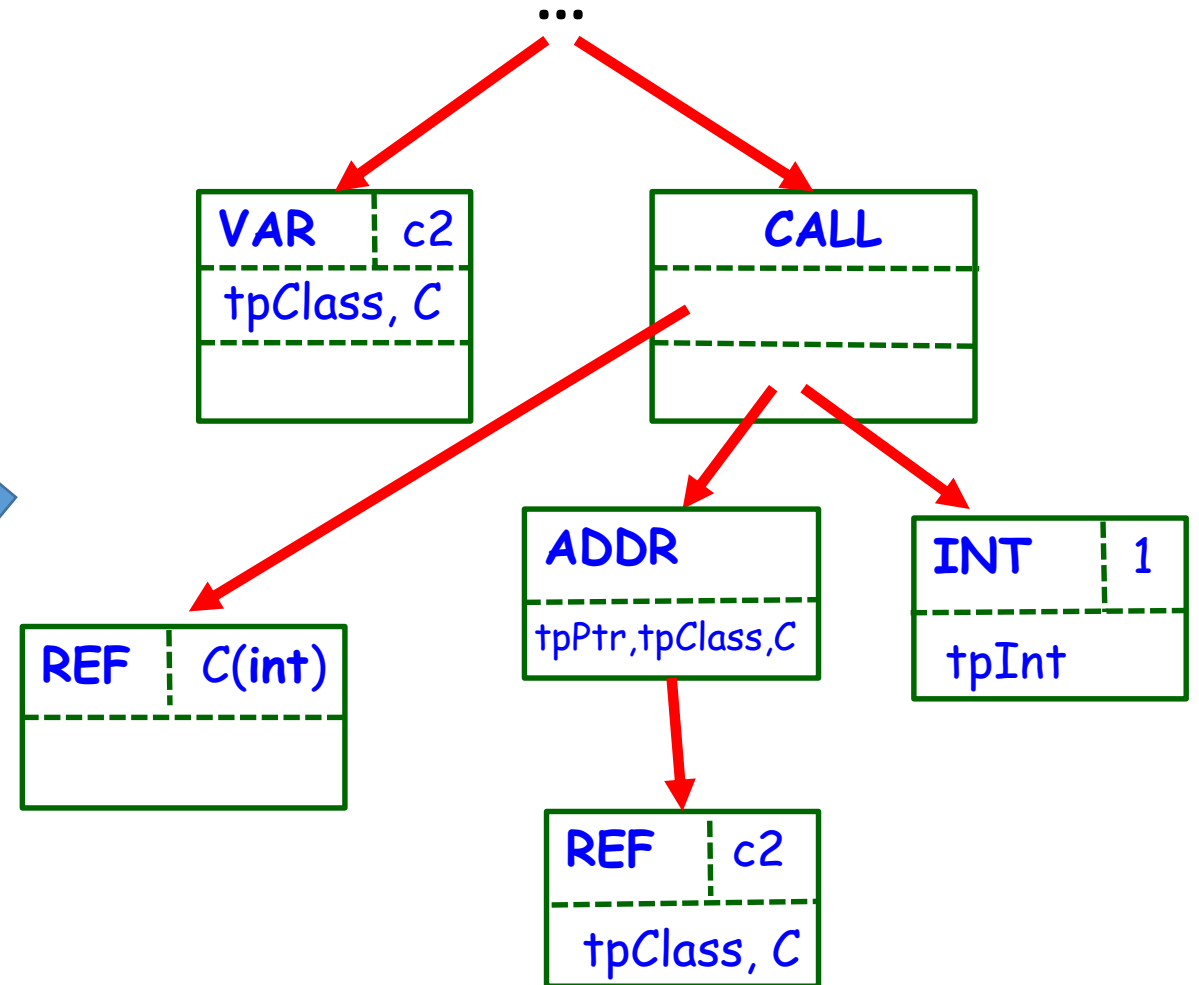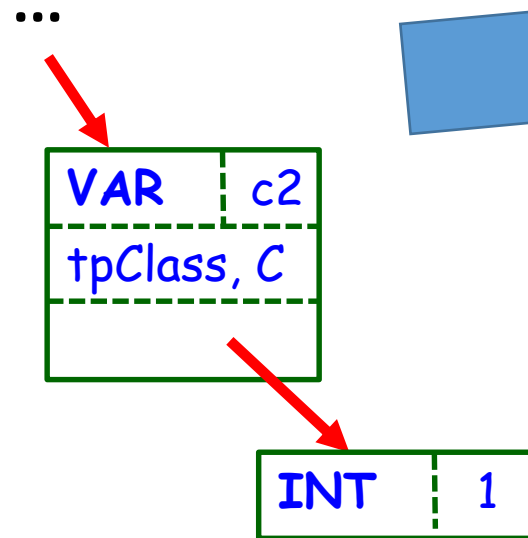
```
            IF
  ┌─────────┬─────────┐
  │         ┆         │
  └ ─ ─ ─ ─ ┴ ─ ─ ─ ─ ┘

        REF ┆ c
  ─ ─ ─ ─ ─ ─ ─ ─ ─ ─
      tpClass,C
```

if (c) …

if ((bool)c) …

Semantic Analysis: Example 4
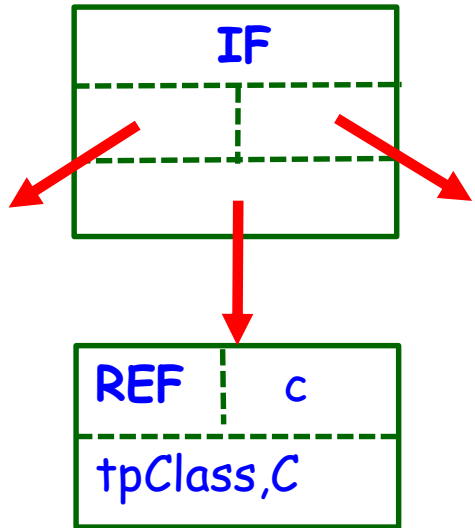
if (c) ...

if ((bool)c) ...

if (c.**operator bool()**) ...

if (C::**operator bool(&c)**) ...

IF

REF | c
tpClass,C

IF

CONV
tpBool

REF | c
tpClass,C

IF

CALL
tpBool

ADDR
tpPtr,tpClass,C

REF | C::**operator bool()()**

REF | c
tpClass,C
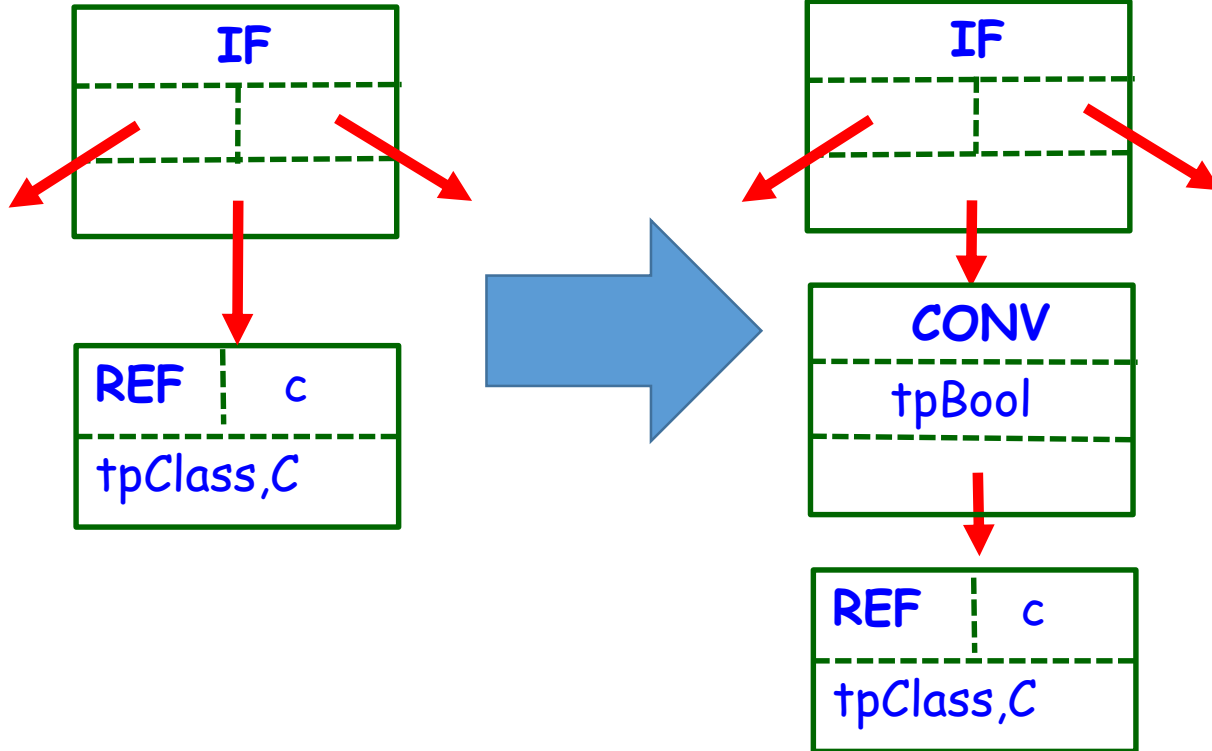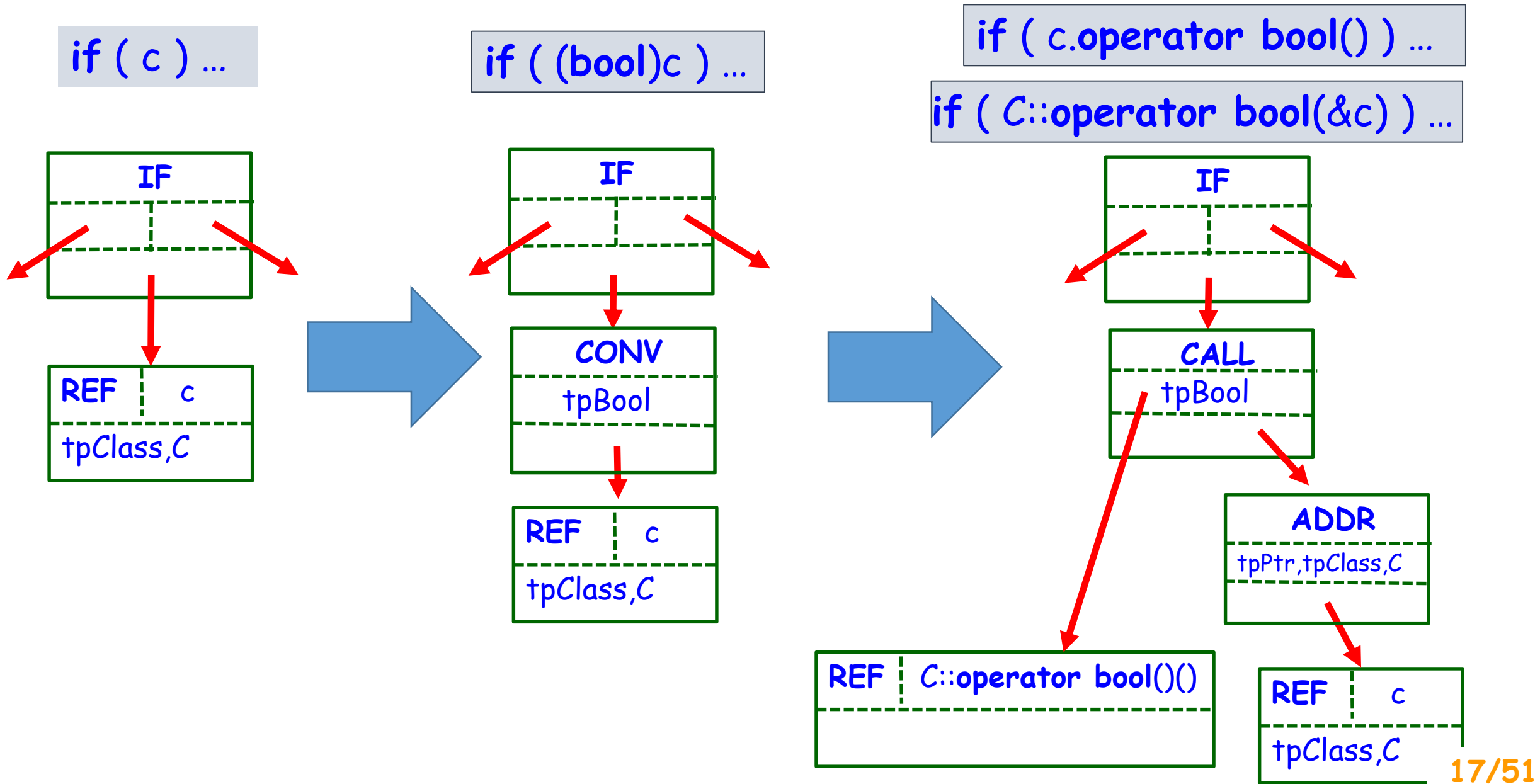
Functional objects ("functors")

```
class C {
public:
    int operator()(int a, int b)
    { return a+b; }
};
...
C c1;

...
int res = c1(5,7);
```
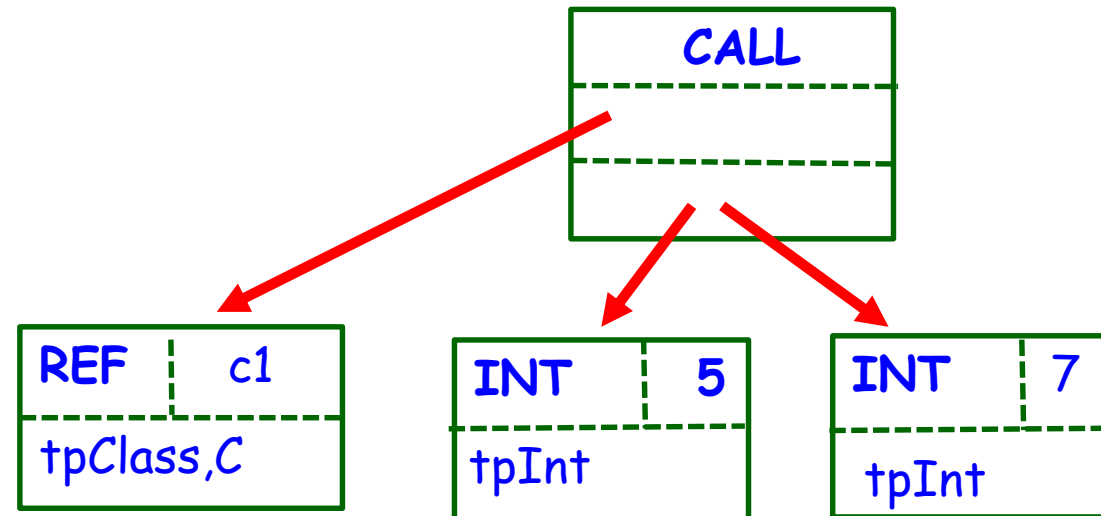

CALL

REF	c1
tpClass,C	

INT	5
tpInt	

INT	7
tpInt	

Functional objects ("functors")

…c1(5,7)…

Functional objects ("functors")

...c1(5,7)... ➡ ...C::**operator**()(&c1,5,7)...

Functional objects ("functors")

...c1(5,7)... ➡ ...C::**operator**()(&c1,5,7)...

Hidden semantics: destructor call

```
class C {
    ...
};
void f() {
    C c;

    ...
}
```

Hidden semantics: destructor call

```
class C {
    ...
};
void f() {
    C c;

    ...
}
```

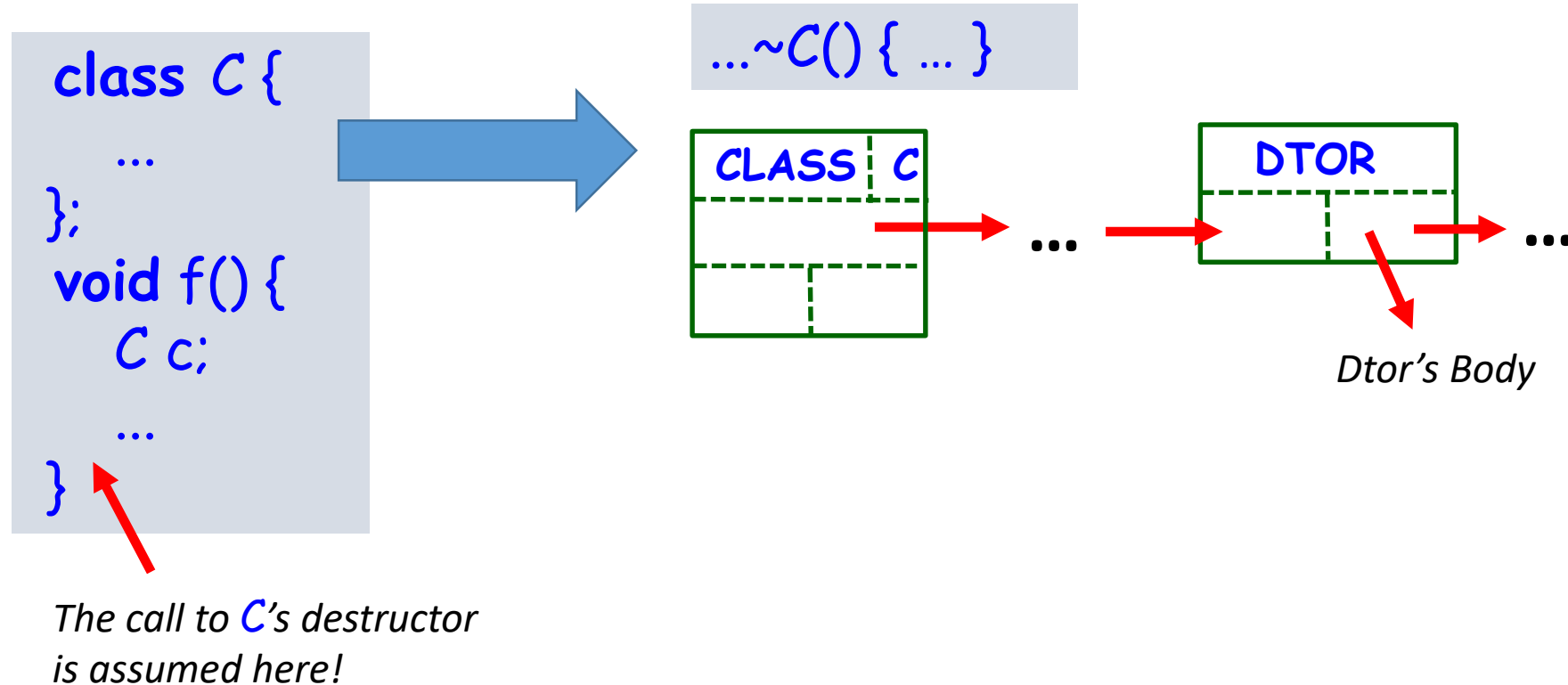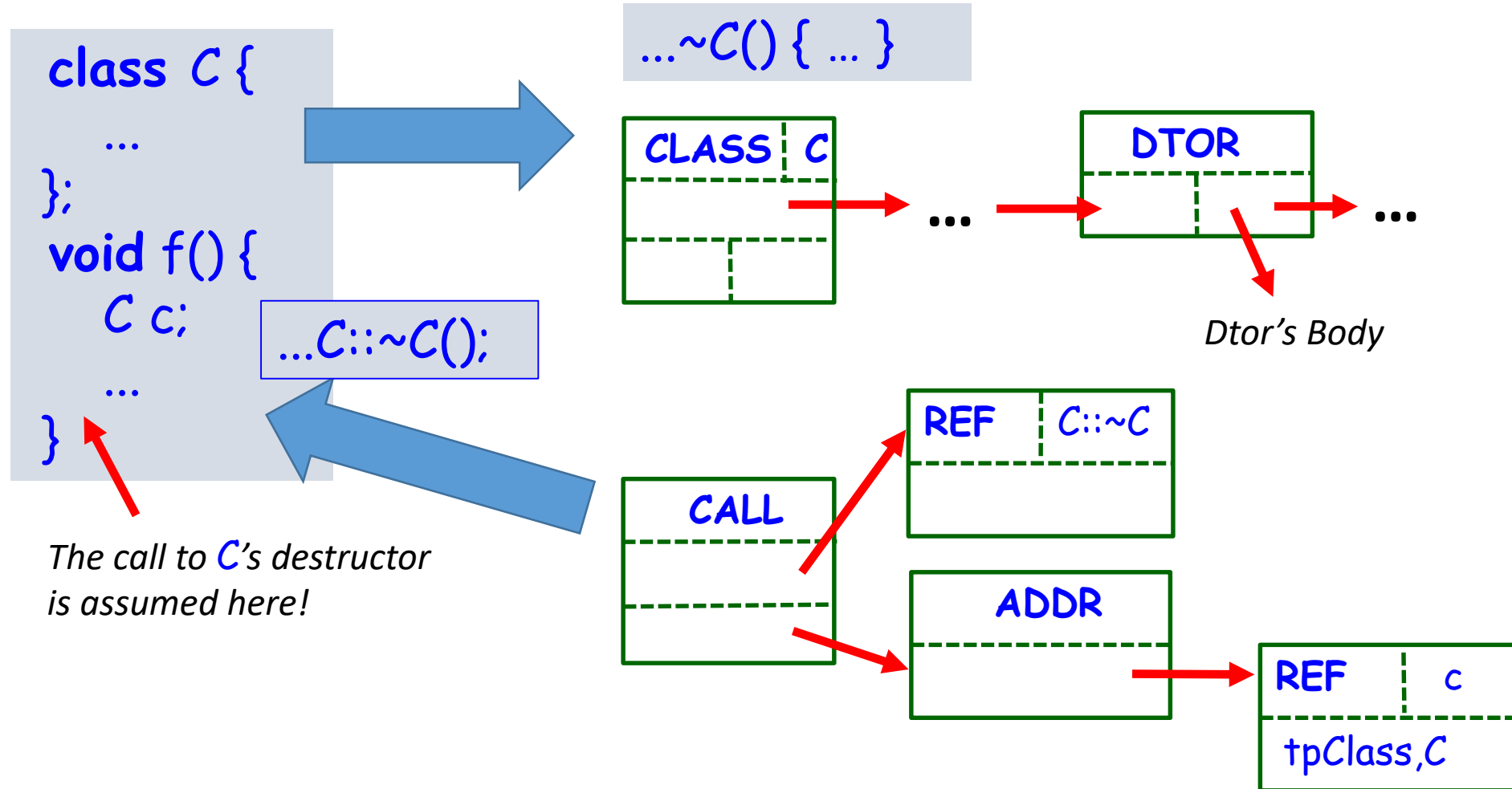*The call to C's destructor is assumed here!*
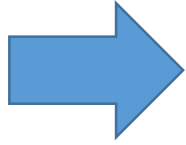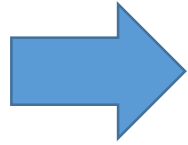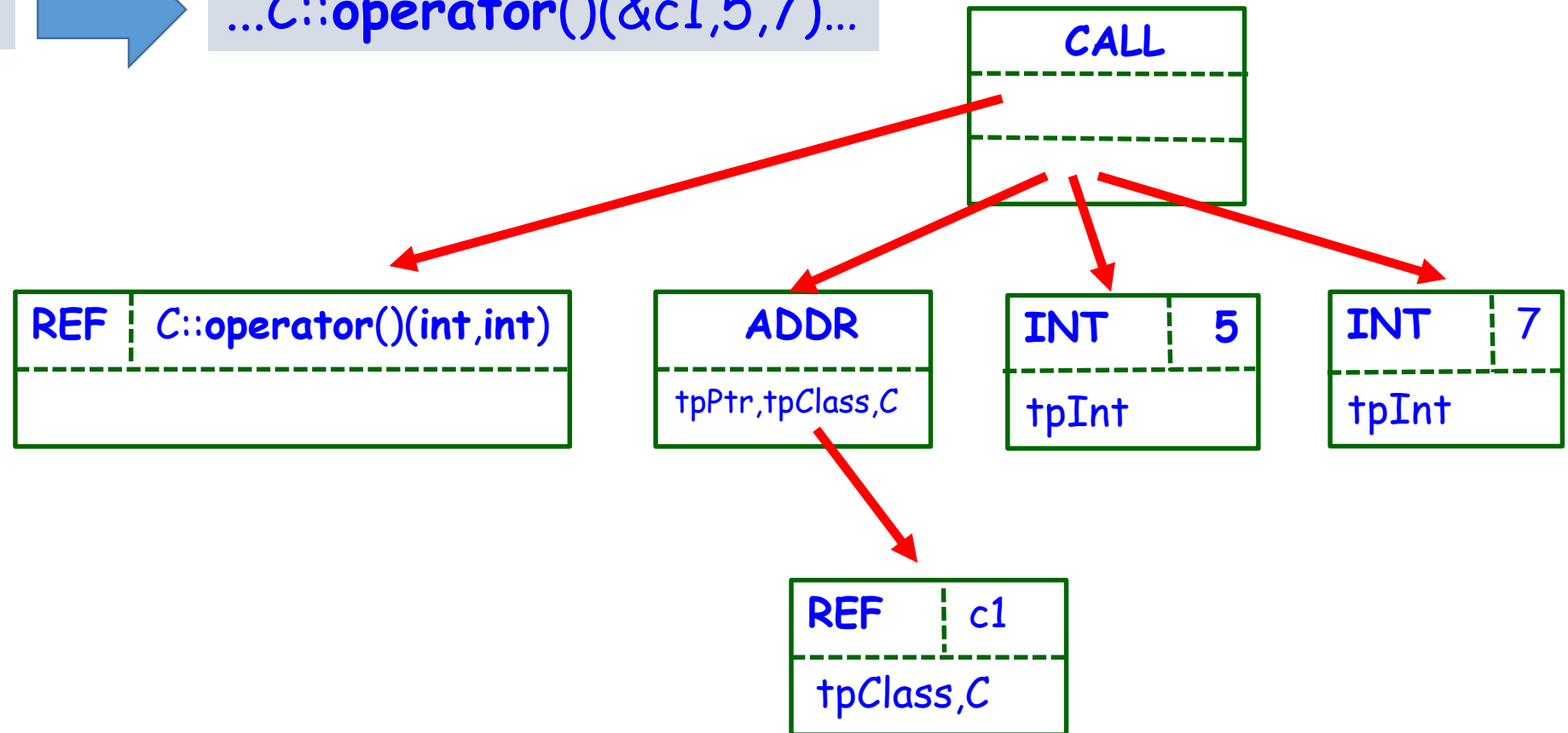
Semantic Analysis: Example 6

Hidden semantics: destructor call

The call to C's destructor is assumed here!

Semantic Analysis: Example 7

Calculating constant expressions

const int Factor = 10;
int A[Factor*7-1];