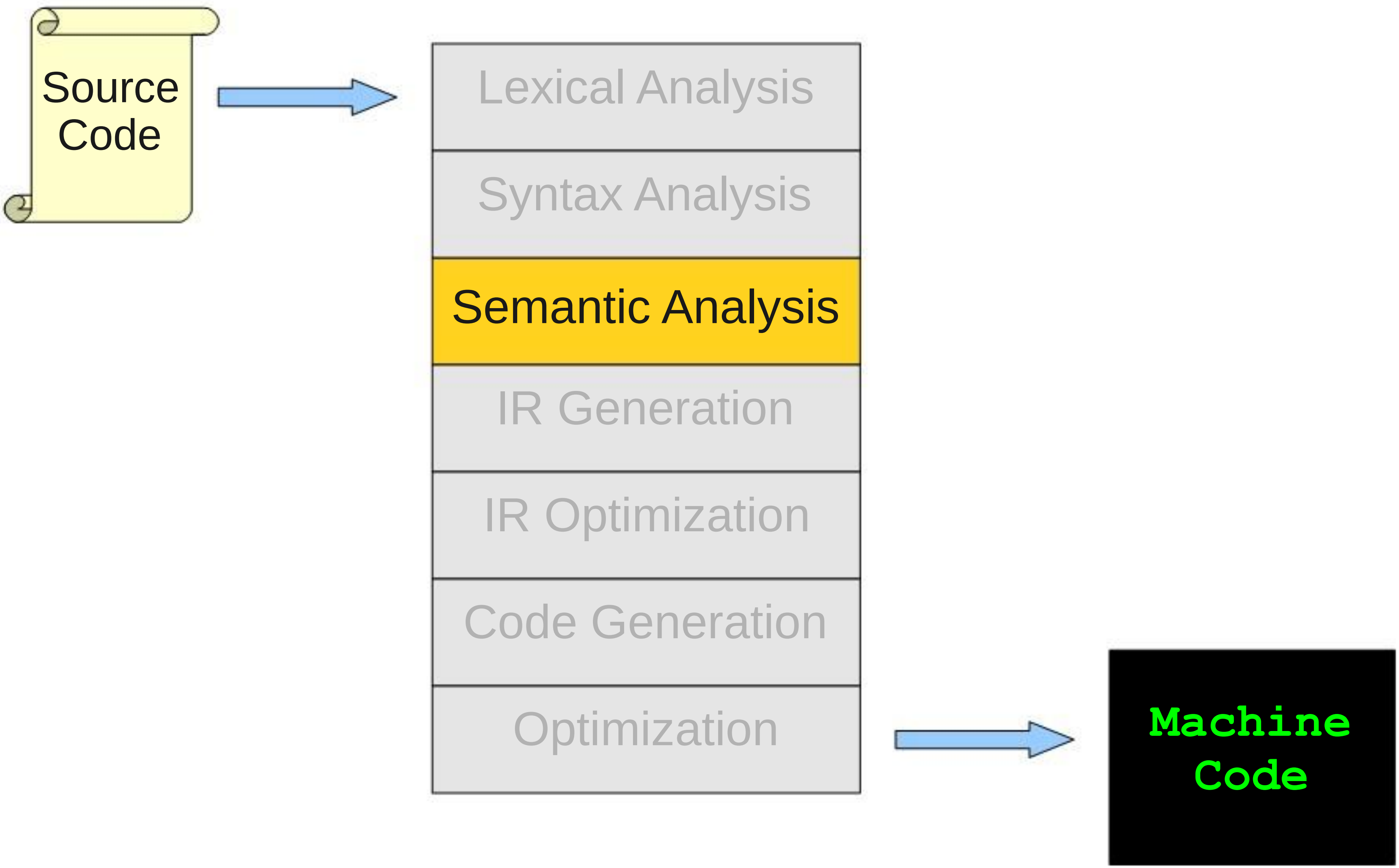
Type Checking

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Type-Checking

- **Type errors** arise when operations are performed on values that do not support that operation.

- **Static type checking.**
  - Analyze the program during compile-time to prove the type of values. E.g. C, C++, Java
  - Explicitly declare type at compile time. Never let bad things happen at runtime.

Ex: int x;
x=1;
{ float x; x =2.0 }; // most-closely nested rule
x= 0;

Type-Checking

- **Dynamic type checking.**
  - Check operations at runtime before performing them.
  - More flexible than static type checking, but usually less efficient.

  E.g. LISP, Python, Perl

Ex:

```
Class A { … }
Class B inherits Class A { … }
Class Main {  Class A   x ; // type of x is A
          if(Condition==True)
                x = new A;   // dynamic type of x is A
          else
                x = new B;   // dynamic type of x is B
}
```

Type Systems

- **Strong type system** is a system that require the programmer to do an explicit type conversion (casting).
  - Java, C++, C#, Python, etc.
  - There is no way you can add a number to a string without doing an explicit conversion.

- **Weak type system** is a system that do automatic type conversion.
  - Visual BASIC, JavaScript, Perl, PHP
  - You can do things like adding numbers to strings and the language will do an implicit coercion for you.

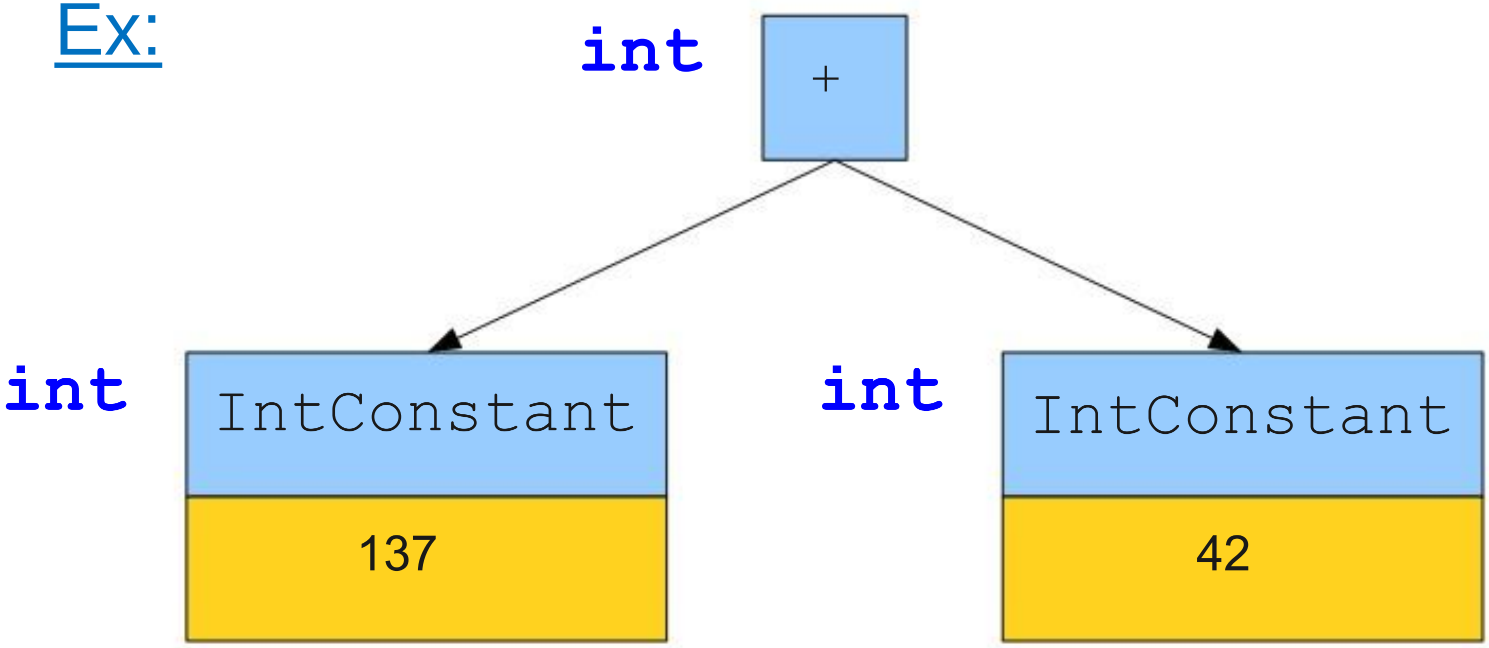
Type Wars

- Endless debate about what the “right” system is.
- **Dynamic type** systems make it easier to prototype; **static type** systems have fewer bugs.
- **Strongly-typed** languages are more efficient, **weakly-typed** systems are less effort on programming.

Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.

Ex:



Sample Inference Rules

- "If **x** is an identifier that refers to an object of type **t**, then **x** has type **t**."
- "If **e** is an integer constant, then **e** has type **`int`**."
- "If the operands e_1 and e_2 of $\mathbf{e_1 + e_2}$ are known to have types **`int`** and **`int`**, then $\mathbf{e_1 + e_2}$ has type **`int`**."

Type Checking as Proofs

- We begin with a set of **axioms**, then apply our **inference rules** to determine the types of expressions.

- We will encode our axioms and inference rules using this syntax:

$$\frac{\text{Preconditions}}{\text{Postconditions}}$$

- This is read “if **preconditions** are true, we can infer **postconditions**.”

Formal Notation for Type Systems

- We write

$$\vdash \mathbf{e} : \mathbf{T}$$

  if the expression **e** has type **T**.

- The symbol $\vdash$ means "we can infer..."

Our Starting Axioms

$$\frac{}{\vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{bool}}$$

Without any hypotheses, it is provable that both **true** and **false** constants have type Boolean.

Some Simple Inference Rules

$$\frac{i \text{ is an integer constant}}{\vdash i : \texttt{int}}$$

$$\frac{s \text{ is a string constant}}{\vdash s : \texttt{string}}$$

$$\frac{d \text{ is a double constant}}{\vdash d : \texttt{double}}$$

$$\frac{e : bool}{\vdash \mathrm{!e} : \texttt{bool}}$$

Negation of e

More Complex Inference Rules

If we can show that e_1 and e_2 have type int…

$$\frac{\vdash e_1 : \mathtt{int} \quad \vdash e_2 : \mathtt{int}}{\vdash e_1 + e_2 : \mathtt{int}}$$

$$\frac{\vdash e_1 : \mathtt{double} \quad \vdash e_2 : \mathtt{double}}{\vdash e_1 + e_2 : \mathtt{double}}$$

... then we can show that $e_1 + e_2$ has type int as well

Even More Complex Inference Rules

$$\frac{\vdash e_1 : T \quad \vdash e_2 : T \quad T \text{ is a primitive type}}{\vdash e_1 \texttt{ == } e_2 : \texttt{bool}}$$

$$\frac{\vdash e_1 : T \quad \vdash e_2 : T \quad T \text{ is a primitive type}}{\vdash e_1 \texttt{ != } e_2 : \texttt{bool}}$$

Why Specify Types this Way?

- Gives a **rigorous definition of types** independent of any particular implementation.

- Gives **maximum flexibility in implementation**.

- Allows formal verification of program properties. This is what's **used in the literature**.

Strengthening our Inference Rules

- Since the same object may have different types in different scopes, we need to strengthen our inference rules to remember **under what circumstances the inference rules are valid**.

- We write $$S \vdash \mathbf{e} : \mathbf{T}$$

  if **in scope S**, the expression **e** has type **T**.

- Types are now proven relative to the scope they are in.

Old Rules Revisited

$$\frac{}{S \vdash \mathtt{true} : \mathtt{bool}}$$

$$\frac{}{S \vdash \mathtt{false} : \mathtt{bool}}$$

$$\frac{i \text{ is an integer constant}}{S \vdash i : \mathtt{int}}$$

$$\frac{s \text{ is a string constant}}{S \vdash s : \mathtt{string}}$$

$$\frac{d \text{ is a double constant}}{S \vdash d : \mathtt{double}}$$

$$\frac{S \vdash e_1 : \mathtt{double} \quad S \vdash e_2 : \mathtt{double}}{S \vdash e_1 + e_2 : \mathtt{double}}$$

$$\frac{S \vdash e_1 : \mathtt{int} \quad S \vdash e_2 : \mathtt{int}}{S \vdash e_1 + e_2 : \mathtt{int}}$$

Rule for Identifier

$$\frac{x \text{ is an identifier.} \quad \boldsymbol{x} \textbf{ is a variable in scope S} \text{ with type T.}}{\mathbf{S} \vdash x : \mathrm{T}}$$

Rules for Functions

f is an identifier.

f is a function in scope S.

f has type $(T_1, \ldots, T_n) \rightarrow U$

$S \vdash e_i : T_i$ for $1 \leq i \leq n$

$$\frac{}{S \vdash f(e_1, \ldots, e_n) : U}$$

Rules for Arrays

$$\frac{S \vdash e_1 : T\texttt{[]} \qquad S \vdash e_2 : \texttt{int}}{S \vdash e_1[e_2] : T}$$

Array of type

Index of array must have type int

Type of an element in an array

Rule for Assignment

$$\frac{\begin{array}{c} S \vdash e_1 : T \\ S \vdash e_2 : T \end{array}}{S \vdash e_1 = e_2 : T}$$

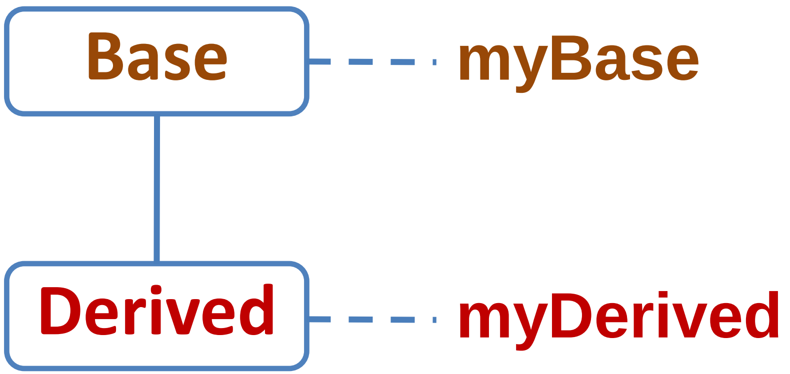
Rule for Assignment

$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T}{S \vdash e_1 = e_2 : T}$$

Problem

If **`Derived`** extends **`Base`**, will this rule work for this code?

```
Base    myBase;
Derived myDerived;
myBase = myDerived;
```



- Some information in myDerived will be thrown away after the assignment statement.
- We need partial orders to solve this problem.

Properties of Inheritance Structures

- Any class is convertible to itself. **(Reflexivity)**
- If A is convertible to B and B is convertible to C, then A is convertible to C. **(Transitivity)**
- If A is convertible to B and B is convertible to A, then A and B are the same type. **(Antisymmetry)**
- The above properties define the **partial orders** over types.

- Base class has more objects, but fewer data and function members

- When we say the class is bigger, it means the class has more data and function members, i.e., Derived class is bigger than Base class.

Types and Partial Orders

- We say that A ≤ B if A is convertible to B.
- We have that
  - A ≤ A
  - A ≤ B and B ≤ C implies A ≤ C
  - A ≤ B and B ≤ A implies A = B

Types and Partial Orders

- Bigger type must be explicitly convertible to Small type. However, there might be a lost in conversion.

  Ex: double x = 1234.7; int a;

  a = (int)x; // Cast double into int.

  In OOP, childclass (bigger) ≤ parentclass (smaller), not vice versa.

- There are pairs of types (/classes) that can not be convertible (neither implicit nor explicit). E.g. In C#, string is not convertible with Integer.

Updated Rule for Assignment

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_2 \leq T_1 \end{array}}{S \vdash e_1 = e_2 : T_1}$$

Updated Rule for Comparisons

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 \;\texttt{==}\; e_2 : \texttt{bool}}$$

$$\Rightarrow$$

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \text{ and } T_2 \text{ are of class type.} \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash e_1 \;\texttt{==}\; e_2 : \texttt{bool}}$$

This makes the comparison between int and string impossible.

Example: Type Inference

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* … */
}
```

Something is wrong here !

Example: Type Inference

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* … */
}
```

$$\frac{x \text{ is an identifier.} \quad x \text{ is a variable in scope S with type T.}}{S \vdash x : T}$$

Facts
$S \vdash$ **x : int**
$S \vdash$ **y : int**
$S \vdash$ **z : int**

Example: Type Inference

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
      /* … */
}
```

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash e_1 \mathbin{\texttt{==}} e_2 : \texttt{bool}}$$

Facts
$S \vdash$ **`x`** : **`int`**
$S \vdash$ **`y`** : **`int`**
$S \vdash$ **`z`** : **`int`**
$S \vdash$ **`x == y`** : **`bool`**

Example: Type Inference

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* … */
}
```

$$\frac{i \text{ is an integer constant}}{S \vdash i : \texttt{int}}$$

Facts
$S \vdash$ **x : int**
$S \vdash$ **y : int**
$S \vdash$ **z : int**
$S \vdash$ **x == y : bool**
$S \vdash$ **5 : int**

Example: Type Inference

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* … */
}
```

$$\frac{S \vdash e_1 : \texttt{int} \qquad S \vdash e_2 : \texttt{int}}{S \vdash e_1 + e_2 : \texttt{int}}$$

Facts
$S \vdash$ **x** : **int**
$S \vdash$ **y** : **int**
$S \vdash$ **z** : **int**
$S \vdash$ **x == y** : **bool**
$S \vdash$ **5** : **int**
$S \vdash$ **x + y** : **int**

Example: Type Inference

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* … */
}
```

$$\frac{S \vdash e_1 : \texttt{int} \qquad S \vdash e_2 : \texttt{int}}{S \vdash e_1 < e_2 : \texttt{bool}}$$

Facts
$S \vdash$ **x** : **int**
$S \vdash$ **y** : **int**
$S \vdash$ **z** : **int**
$S \vdash$ **x == y** : **bool**
$S \vdash$ **5** : **int**
$S \vdash$ **x + y** : **int**
$S \vdash$ **x + y < z** : **bool**

Example: Type Inference

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* … */
}
```

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash e_1 == e_2 : \mathbf{bool}}$$

Facts
$S \vdash$ **x : int**
$S \vdash$ **y : int**
$S \vdash$ **z : int**
$S \vdash$ **x == y : bool**
$S \vdash$ **5 : int**
$S \vdash$ **x + y : int**
$S \vdash$ **x + y < z : bool**
$S \vdash$ **x == z : bool**

Example: Type Inference

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* … */
}
```

$$\frac{S \vdash e_1 : \texttt{int} \qquad S \vdash e_2 : \texttt{int}}{S \vdash e_1 > e_2 : \texttt{bool}}$$

Facts
S ⊢ **x : int**
S ⊢ **y : int**
S ⊢ **z : int**
S ⊢ **x == y : bool**
S ⊢ **5 : int**
S ⊢ **x + y : int**
S ⊢ **x + y < z : bool**
S ⊢ **x == z : bool**

Example: Type Inference

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* … */
}
```

$$\frac{S \vdash e_1 : \texttt{int} \quad S \vdash e_2 : \texttt{int}}{S \vdash e_1 > e_2 : \texttt{bool}}$$

```
> Error: Cannot compare int and bool
```

Facts
$S \vdash$ **x : int**
$S \vdash$ **y : int**
$S \vdash$ **z : int**
$S \vdash$ **x == y : bool**
$S \vdash$ **5 : int**
$S \vdash$ **x + y : int**
$S \vdash$ **x + y < z : bool**
$S \vdash$ **x == z : bool**

Example: Type Inference

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
     /* … */
}
```

$$\frac{S \vdash e_1 : \texttt{bool} \qquad S \vdash e_2 : \texttt{bool}}{S \vdash e_1 \;\texttt{\&\&}\; e_2 : \texttt{bool}}$$

```
> Error: Cannot compare int and bool
  Error: Cannot compare ??? and bool
```

Facts
$S \vdash$ **x** : **int**
$S \vdash$ **y** : **int**
$S \vdash$ **z** : **int**
$S \vdash$ **x == y** : **bool**
$S \vdash$ **5** : **int**
$S \vdash$ **x + y** : **int**
$S \vdash$ **x + y < z** : **bool**
$S \vdash$ **x == z** : **bool**

Example: Type Inference

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
    /* … */
}
```

$$\frac{S \vdash e_1 : \texttt{bool} \qquad S \vdash e_2 : \texttt{bool}}{S \vdash e_1 \;||\; e_2 : \texttt{bool}}$$

```
> Error: Cannot compare int and bool
  Error: Cannot compare ??? and bool
  Error: Cannot compare ??? and bool
```

Facts
$S \vdash \texttt{x} : \texttt{int}$
$S \vdash \texttt{y} : \texttt{int}$
$S \vdash \texttt{z} : \texttt{int}$
$S \vdash \texttt{x == y} : \texttt{bool}$
$S \vdash \texttt{5} : \texttt{int}$
$S \vdash \texttt{x + y} : \texttt{int}$
$S \vdash \texttt{x + y < z} : \texttt{bool}$
$S \vdash \texttt{x == z} : \texttt{bool}$

Cascading Errors

- A **type error** occurs when we cannot prove the type of an expression.
- Type errors can easily **cascade**:
  - Can't prove a type for e1, so can't prove a type for e1+e2, so can't prove a type for (e1+e2)+e3, etc.
  - Cascade of type error is annoying. Type error recovery method could be applied.

Cascading Errors

Solution (Type-Error Recovery)

- Define a new type called "NoType" which is a subtype of Boolean type. NoType ≤ Boolean type.
- When we see NoType, we don't produce an error message.

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
```

Boolean Int

$$\frac{S \vdash e_1 : \mathbf{int} \qquad S \vdash e_2 : \mathbf{int}}{S \vdash e_1 > e_2 : \mathbf{bool}}$$

```
> Error: Cannot compare int
         with bool
```

After report an error, we conclude (x==y) > 5 having **NoType**.

Cascading Errors

Solution

- Define a new type called "NoType" which is a subtype of Boolean type. NoType ≤ Boolean type.
- When we see NoType, we don't produce an error message.

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
```

NoType Bool

Cascading Errors

Solution

- Define a new type called "NoType" which is a subtype of Boolean type. NoType ≤ Boolean type.
- When we see NoType, we don't produce an error message.

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
```

$$\frac{S \vdash e_1 : \mathbf{bool} \quad S \vdash e_2 : \mathbf{bool}}{S \vdash e_1 \ \&\& \ e_2 : \mathbf{bool}}$$

Update rule

$$\frac{S \vdash e_1 : \mathbf{T1} \quad S \vdash e_2 : \mathbf{T2} \quad \mathbf{T1}, \mathbf{T2} \leq \mathbf{bool}}{S \vdash e_1 \ \&\& \ e_2 : \mathbf{bool}}$$

Cascading Errors

Solution

- Define a new type called “NoType” which is a subtype of Boolean type. NoType ≤ Boolean type.
- When we see NoType, we don’t produce an error message.

```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
```

Bool **Bool**

Cascading Errors

Solution

- Define a new type called “NoType” which is a subtype of Boolean type. NoType ≤ Boolean type.
- When we see NoType, we don’t produce an error message.

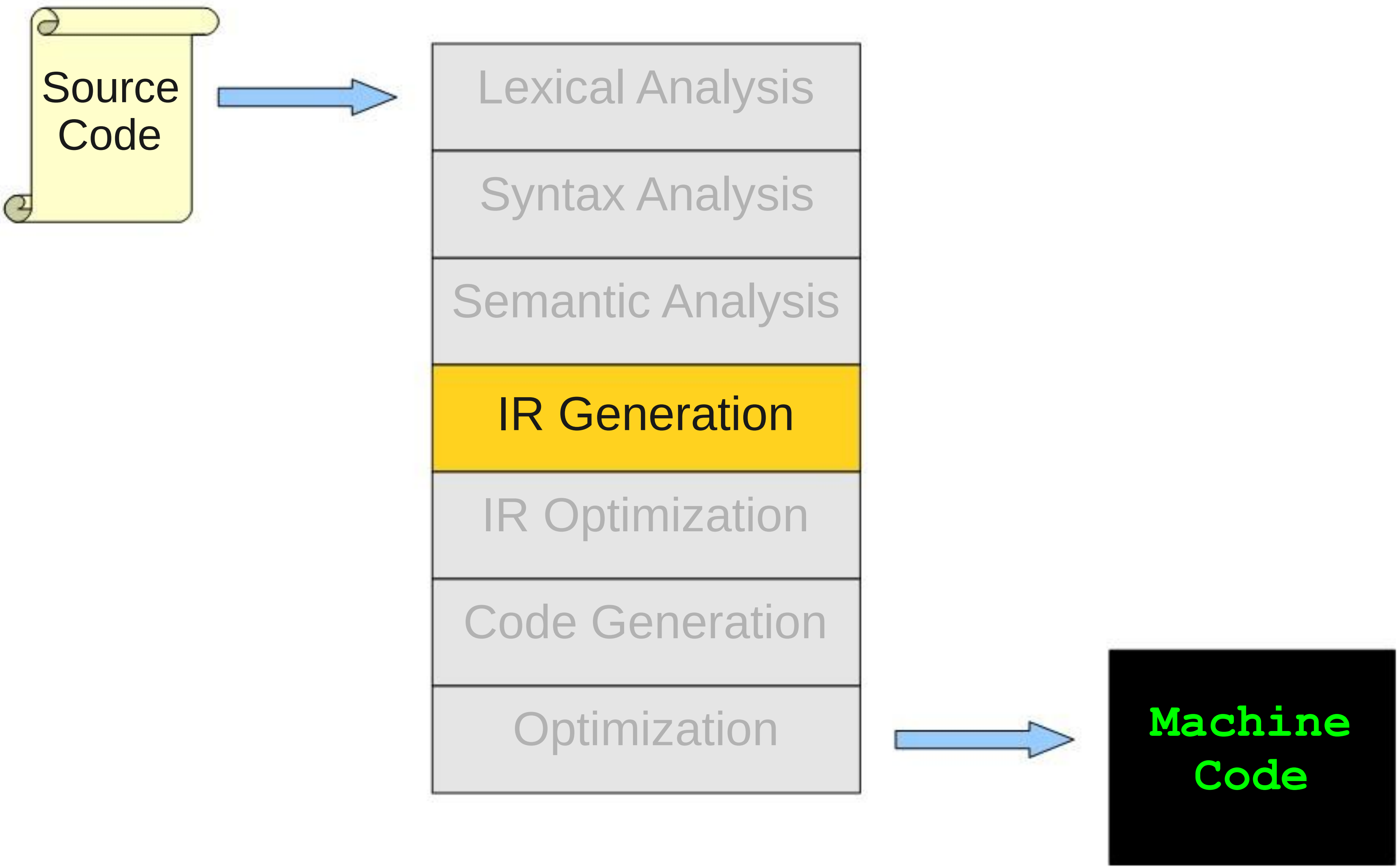
```
int x, y, z;
if (((x == y) > 5 && x + y < z) || x == z) {
```

Bool

No more error message.

IR Generation

Where We Are



Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

What is IR Generation?

- **Intermediate Representation Generation**.
- The final phase of the compiler front-end.
- Goal: Translate the program into the format (assembly-like) expected by the compiler back-end.

Why do IR Generation?

- Working with an intermediate language makes optimizations easier and clearer.

Outline

- **Runtime Environments**
- **Three-Address Code**
- **TAC generation**

Runtime Environment

- A **runtime environment** is a set of data structures maintained at runtime (E.g. the stack, the heap, static data, etc.).

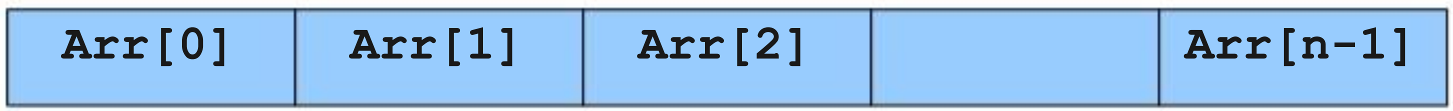
  - What do objects look like in memory?
  - What do functions look like in memory?
  - Where in memory should they be placed?
  - How are function calls implemented?

Encoding Primitive Types

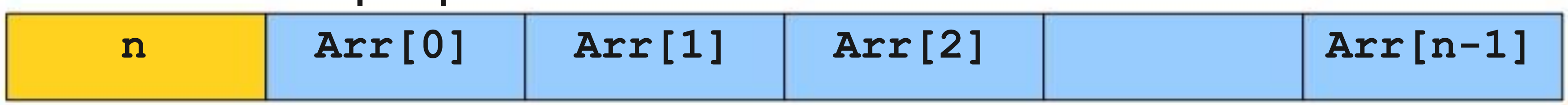
- Primitive **integral** types (**`byte`**, **`char`**, **`short`**, **`int`**, **`long`**, **`unsigned`**, **`uint16_t`**, etc.) typically map directly to the underlying machine type.
- Primitive **real-valued** types (**`float`**, **`double`**, **`long double`**) typically map directly to underlying machine type.
- **Pointers** typically implemented as integers holding memory addresses.

Encoding Arrays

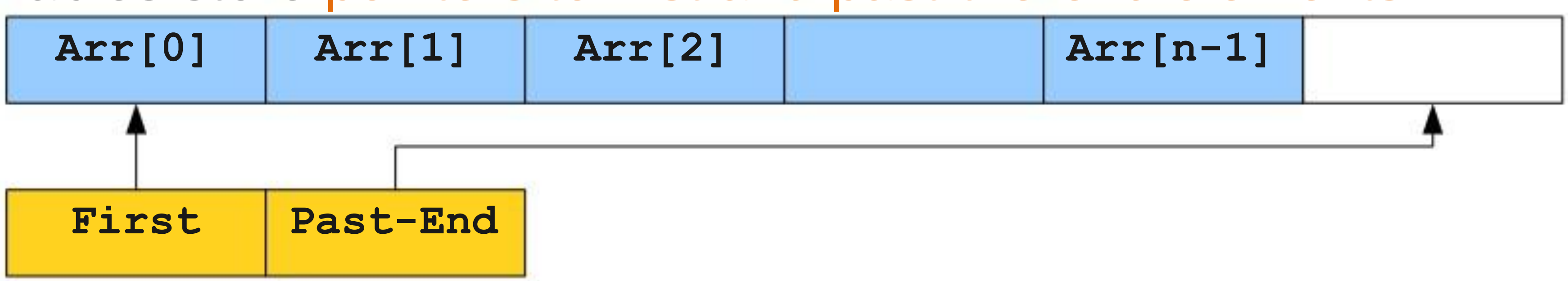
- **C-style** arrays: Elements laid out consecutively in memory.



- **Java-style** arrays: Elements laid out consecutively in memory with size information prepended.



- **D-style** arrays: Elements laid out consecutively in memory; array variables store pointers to first and past-the-end elements.



D is a language with C-like syntax and static typing.(http://dlang.org/index.html)

Encoding Multidimensional Arrays

- Often represented as an array of arrays.

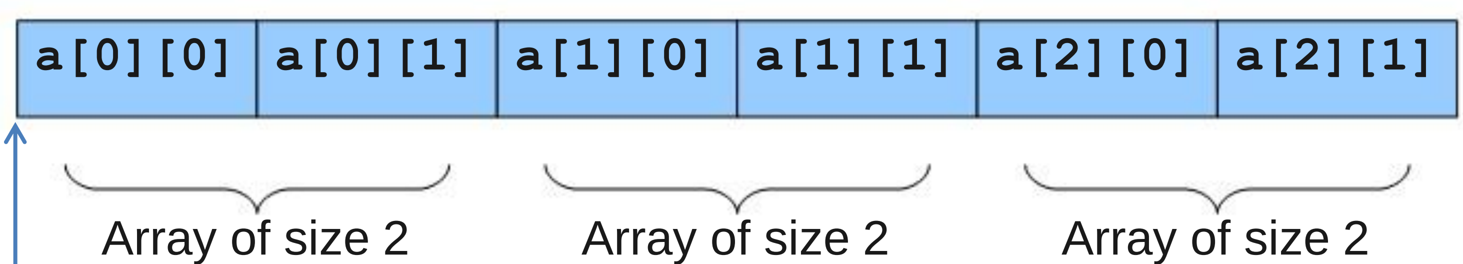
- **C-style** arrays:

`int a[3][2];`

4 Bytes

How do you know an address for an element a[i][j] ?

= base address + (i - 0)*4*2 + (j - 0)*4



Row size

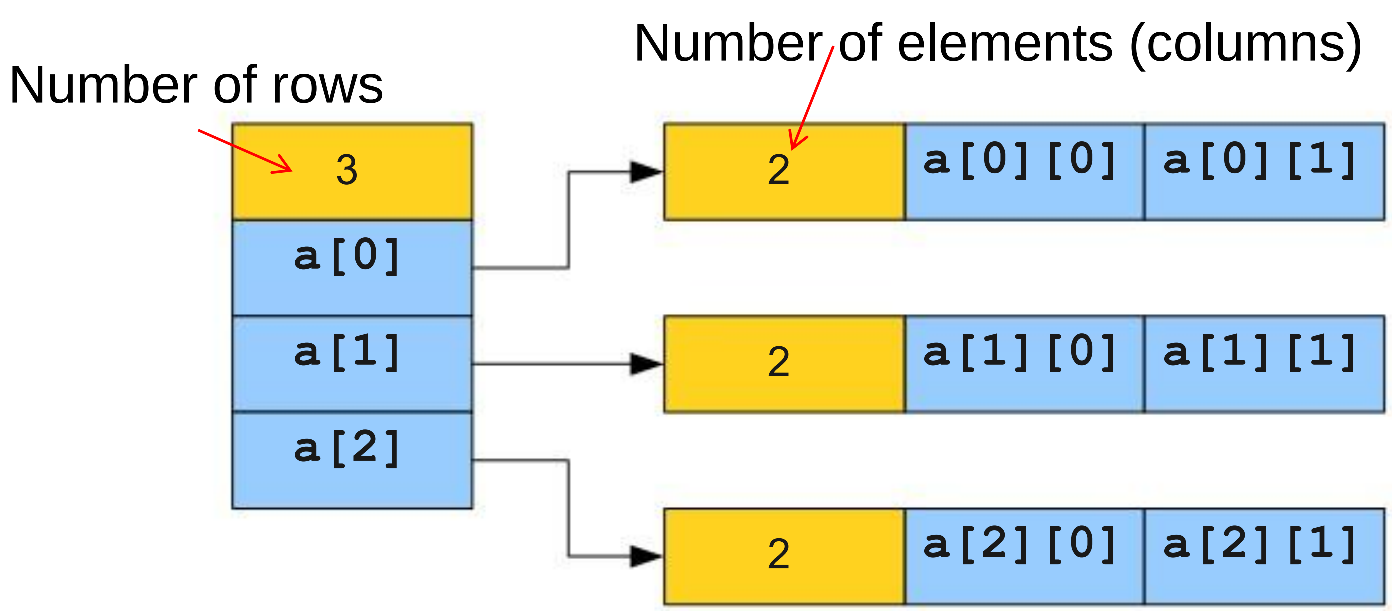
Element size

Base address

Encoding Multidimensional Arrays

- **Java-style** arrays:

```
int[][] a = new int [3][2];
```

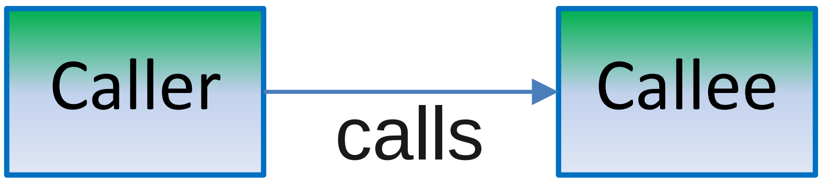


Implementing Function Calls

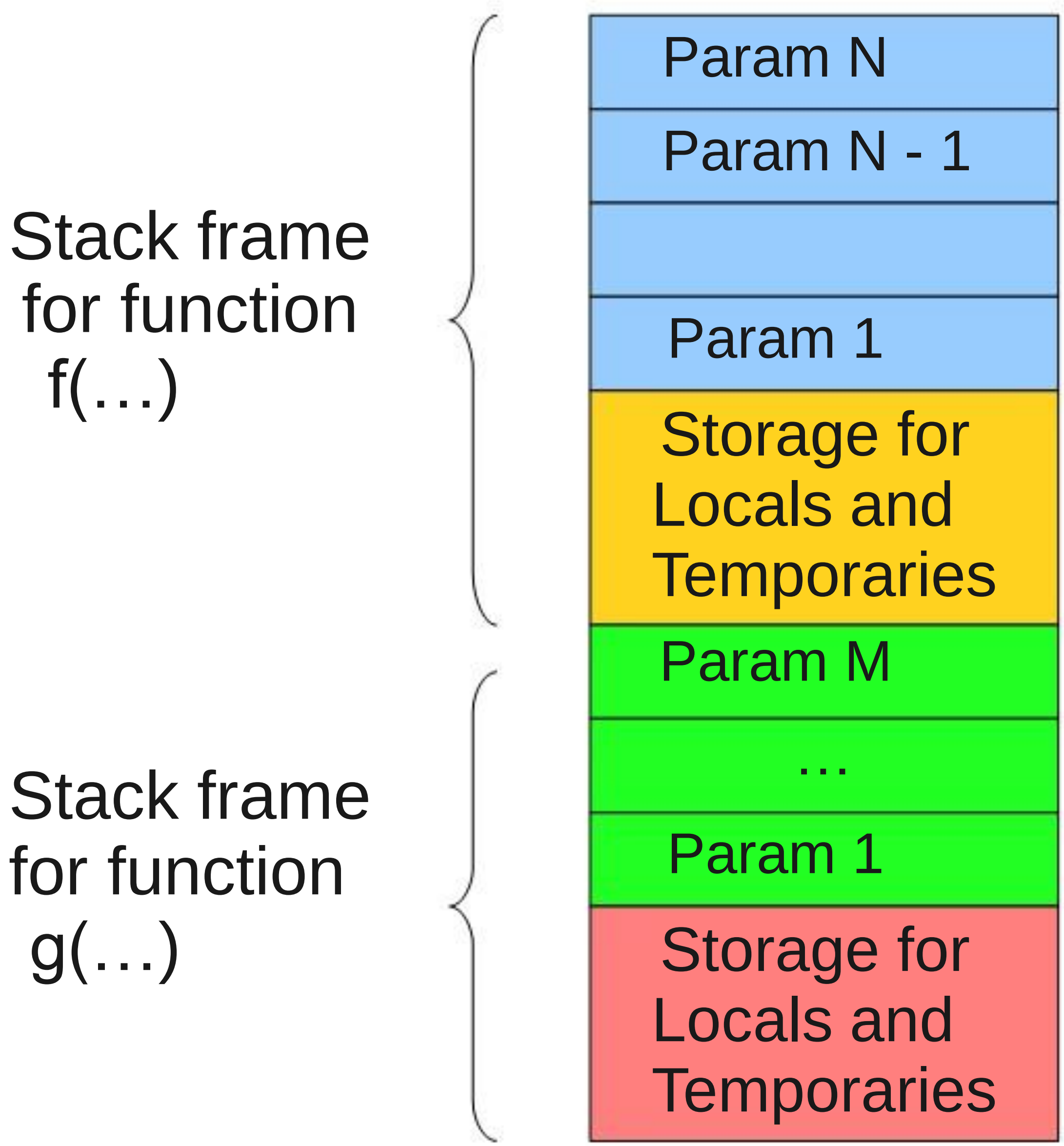
- Function calls are often implemented using a **stack of activation records** (or **stack frames**).
- **Calling** a function **pushes** a new activation record onto the stack.
- **Returning** from a function **pops** the current activation record from the stack.

Implementing Function Calls

- Each **activation record** needs to hold
  - All of its parameters.
  - All of its local variables.
  - All temporary variables.
- Caller responsible for pushing and popping space for callee's arguments.
- Callee responsible for pushing and popping space for its own temporaries.



A Logical Stack Frame
(Logical Activation Record)

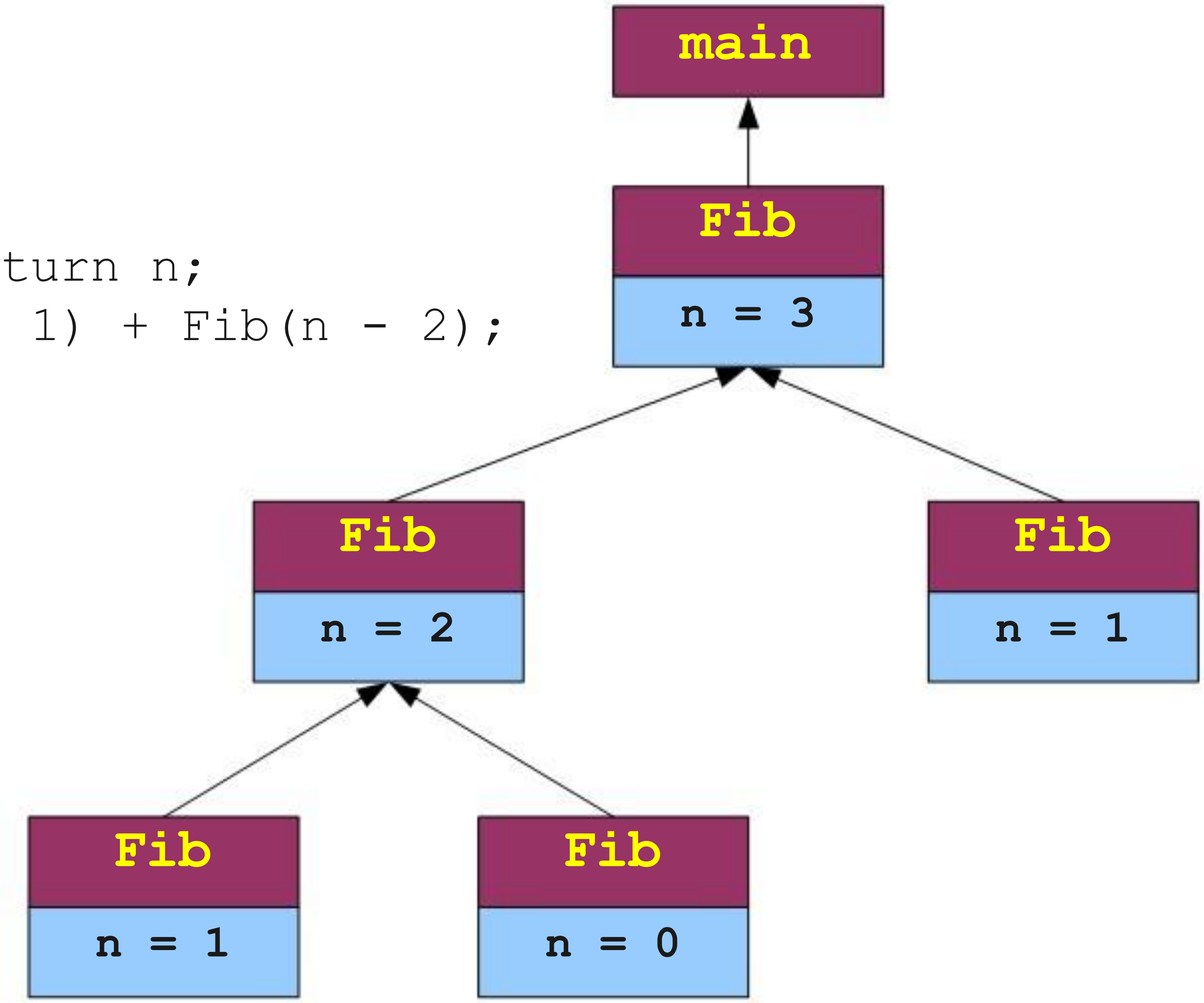


Activation Trees

- An **activation tree** is a tree structure representing all of the function calls made by a program on execution.
- Each **node** in the tree is an **activation record**.
- Each activation record stores a **control link** to the activation record of the function that invoked it.

Activation Trees

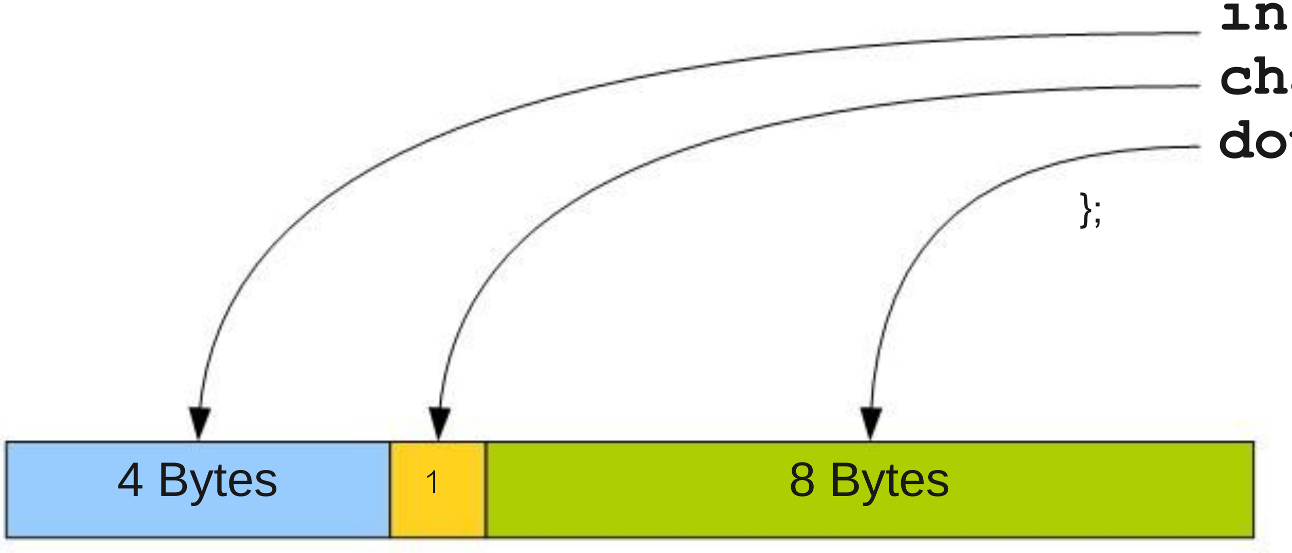
```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```



Encoding C-Style `structs`

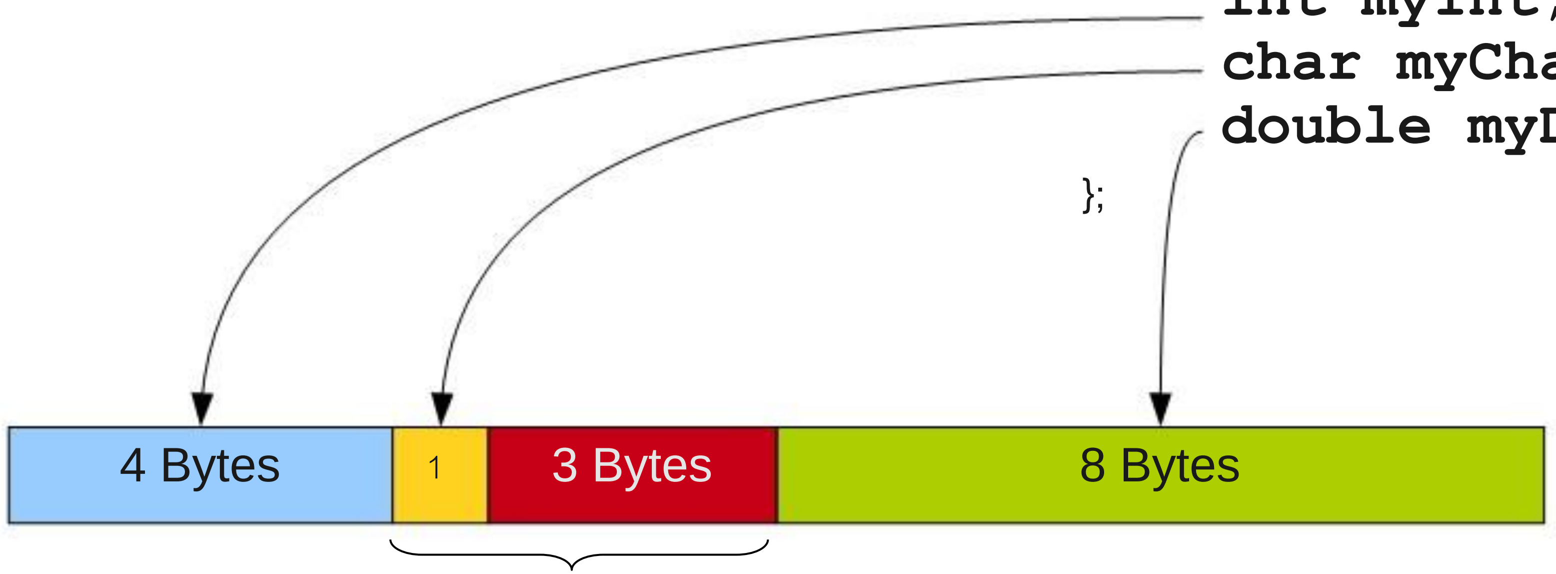
- A `struct` is a type containing a collection of named values.
- Most common approach: lay each field out in the order it's declared.

```
struct MyStruct {
    int myInt;
    char myChar;
    double myDouble;
};
```



Encoding C-Style **`structs`**

```
struct MyStruct {
    int myInt;
    char myChar;
    double myDouble;
};
```



Physical memory layout : Assume that memory is allocated **4-byte/block**

OOP Inheritance

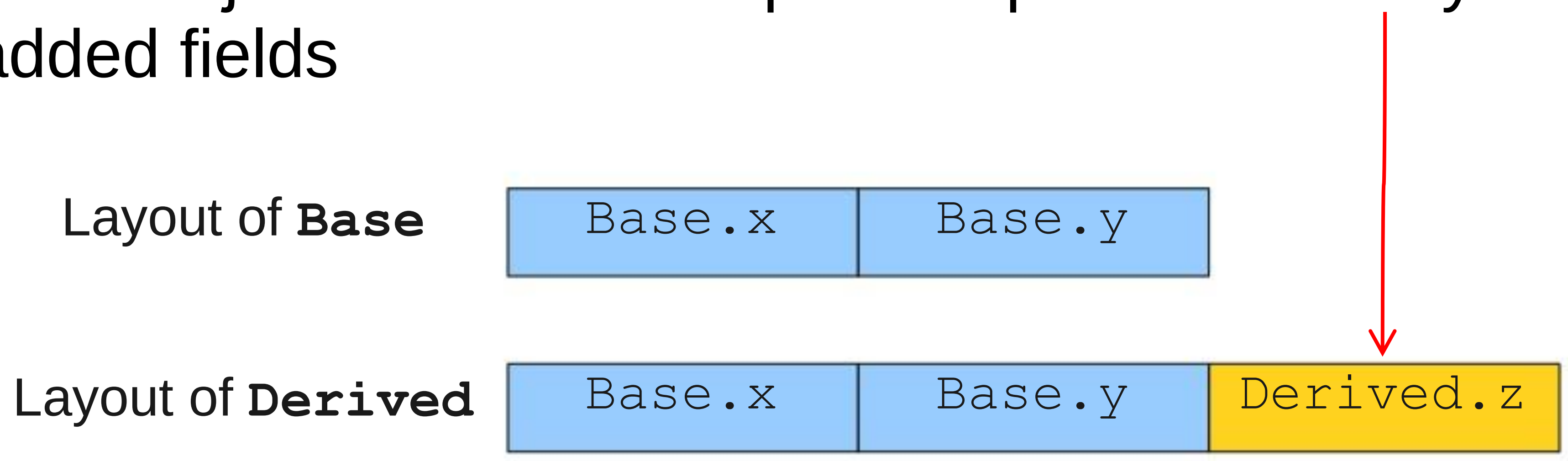
- Consider the following code:

```
class Base {
    int x;
    int y;
}
class Derived extends Base {
    int z;
}
```

- What will `Derived` look like in memory?

An Observation

- Child object has the same memory layout as parent objects.
- Child object needs more space to place the newly added fields



Implementing Dynamic Dispatch

- **Dynamic dispatch** means determining which function to call at runtime based on the dynamic type of the object.

Example:

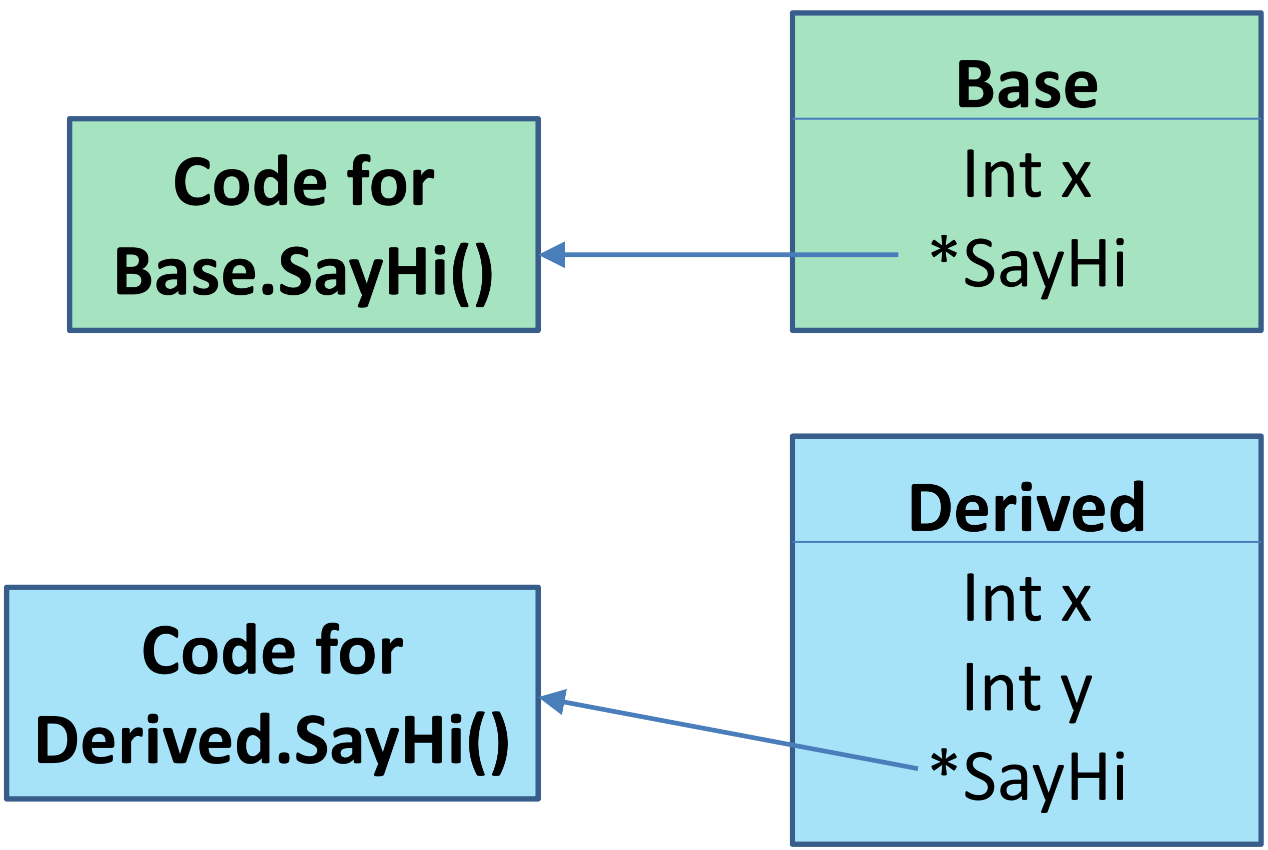
```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
}
```

```
class Derived extends Base {
    int y;
    void sayHi() {
        Print("Derived");
    }
}
```

Virtual function tables can solve the problem.

Virtual Function Tables

- A **virtual function table** (or **vtable**) is an array of pointers to the code of the member function for a particular class.



This is a Pretty Good Idea

- Advantages:
  - Time to determine function to call is O(1).

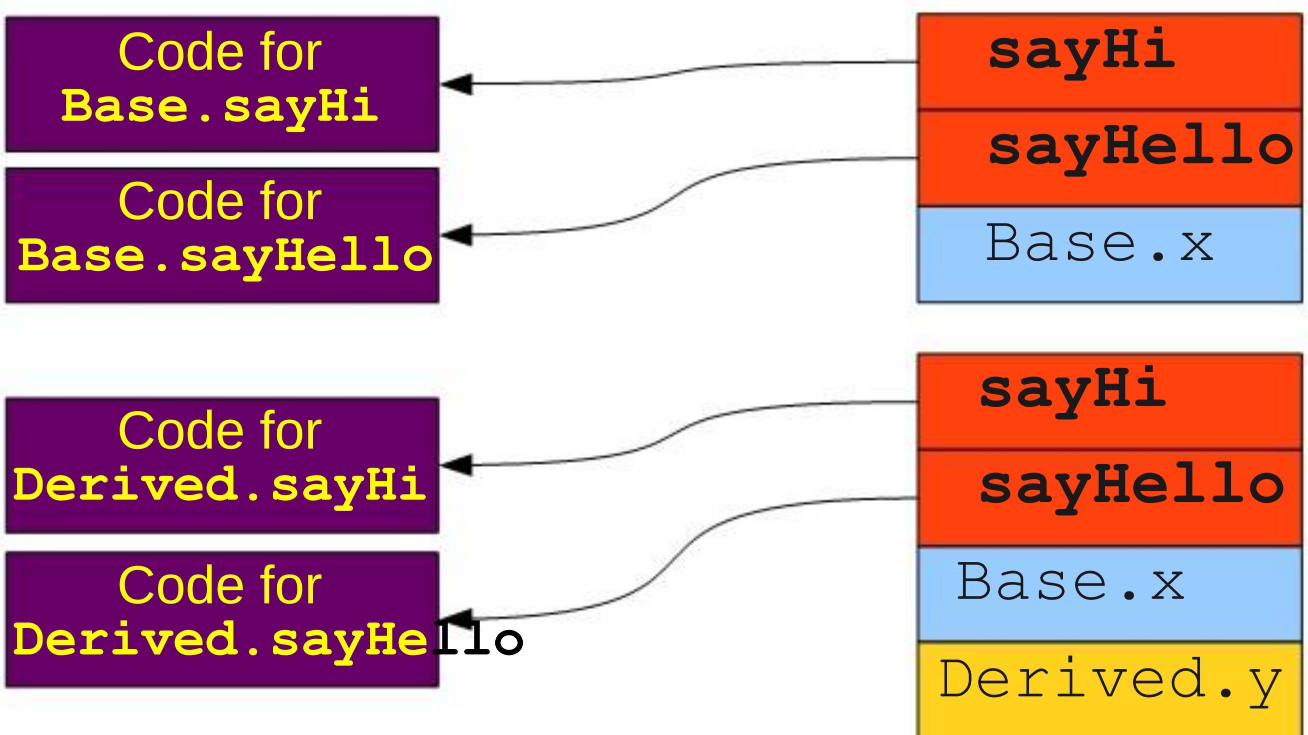
- What are the disadvantages?
  - Each object has its own virtual function table.
  - Wasteful space.

A Common Optimization

```
class Base {
     int x;
     void sayHi() {
          Print("Base1");
     }
     void sayHello() {
          Print(“Base2”);
     }
}
```

```
class Derived extends Base {
     int y;
     void sayHi() {
          Print("Derived1");
     }
     Derived sayHello() {
           Print("Derived2");
     }
}
```

Before optimization

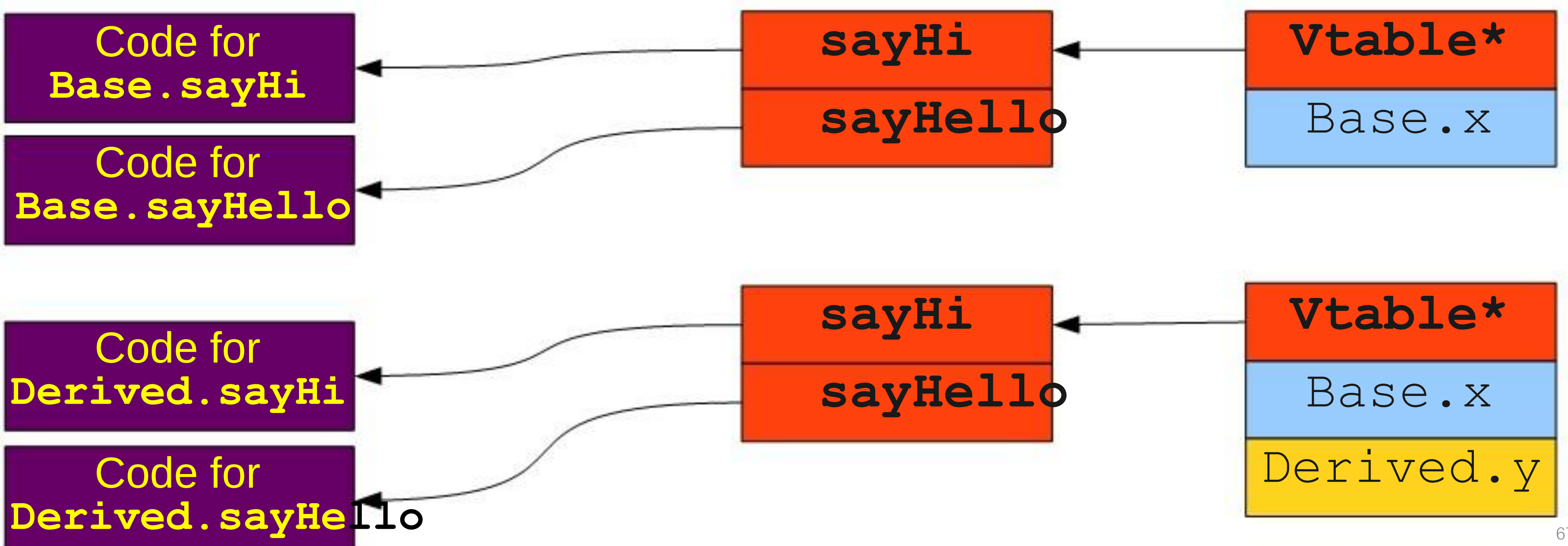


A Common Optimization

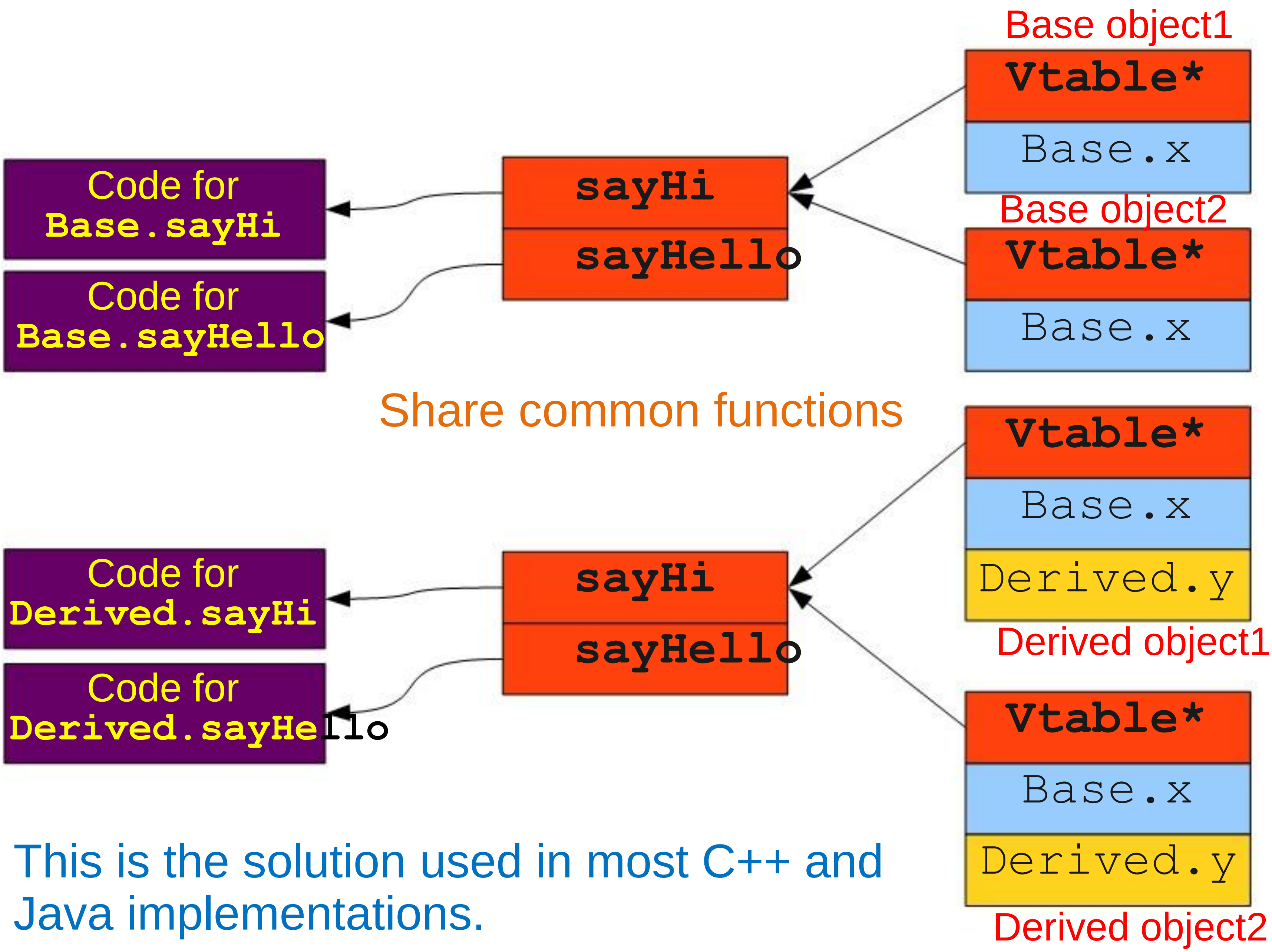
```
class Base {
    int x;
    void sayHi() {
        Print("Base1");
    }
    void sayHello() {
        Print("Base2");
    }
}
```

```
class Derived extends Base {
    int y;
    void sayHi() {
        Print("Derived1");
    }
    Derived sayHello() {
        Print("Derived2");
    }
}
```

After optimization



Objects in Memory



This is the solution used in most C++ and Java implementations.

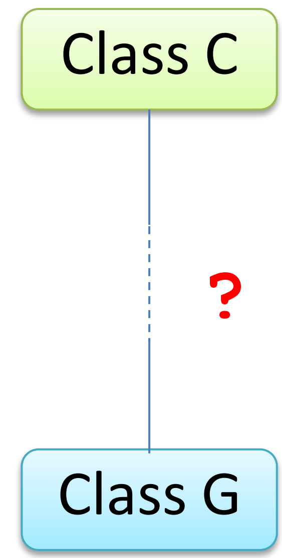
Implementing Dynamic Type Checks

Dynamic Type Checks

- Some languages require dynamic type checking.

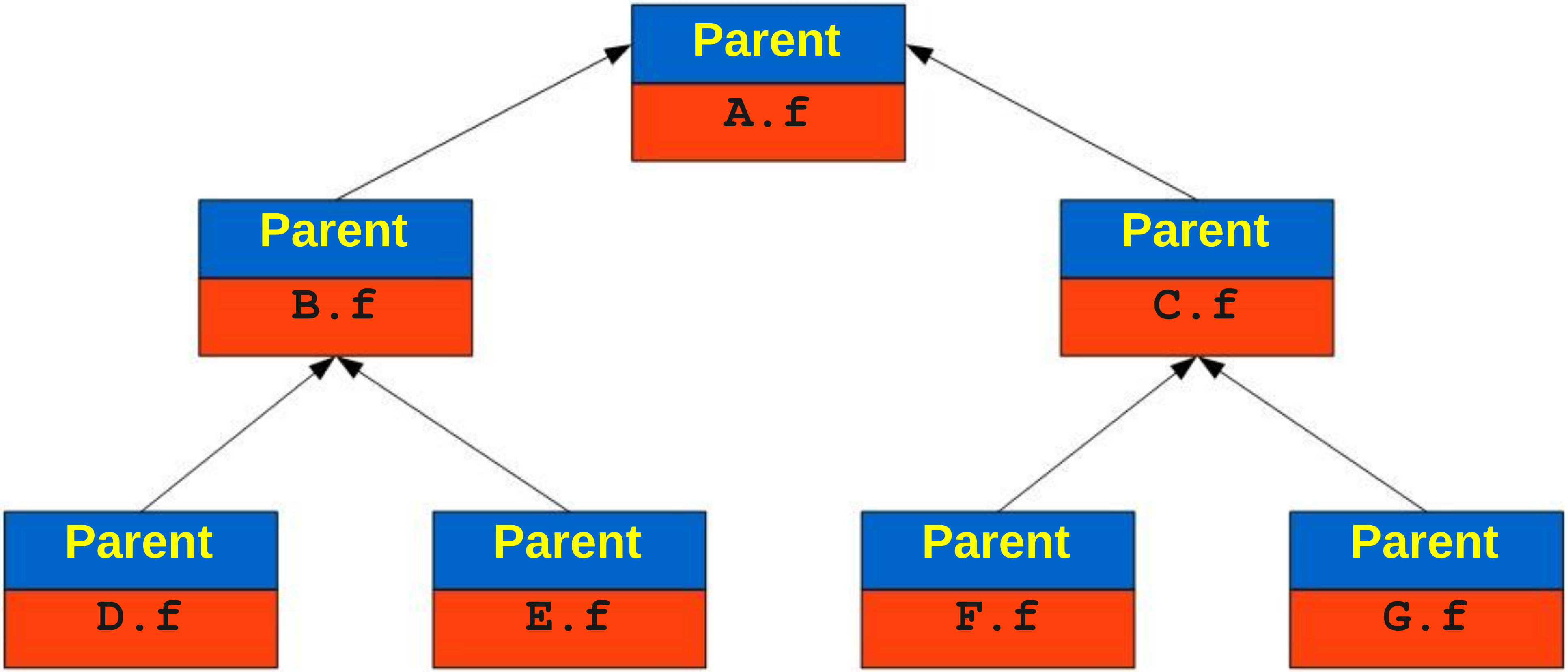
Ex:

```
If(…)
    G myObject = New instance of B
Else
    G myObject = New instance of C

if (myObject instanceof C) {
     /* … */
}
```



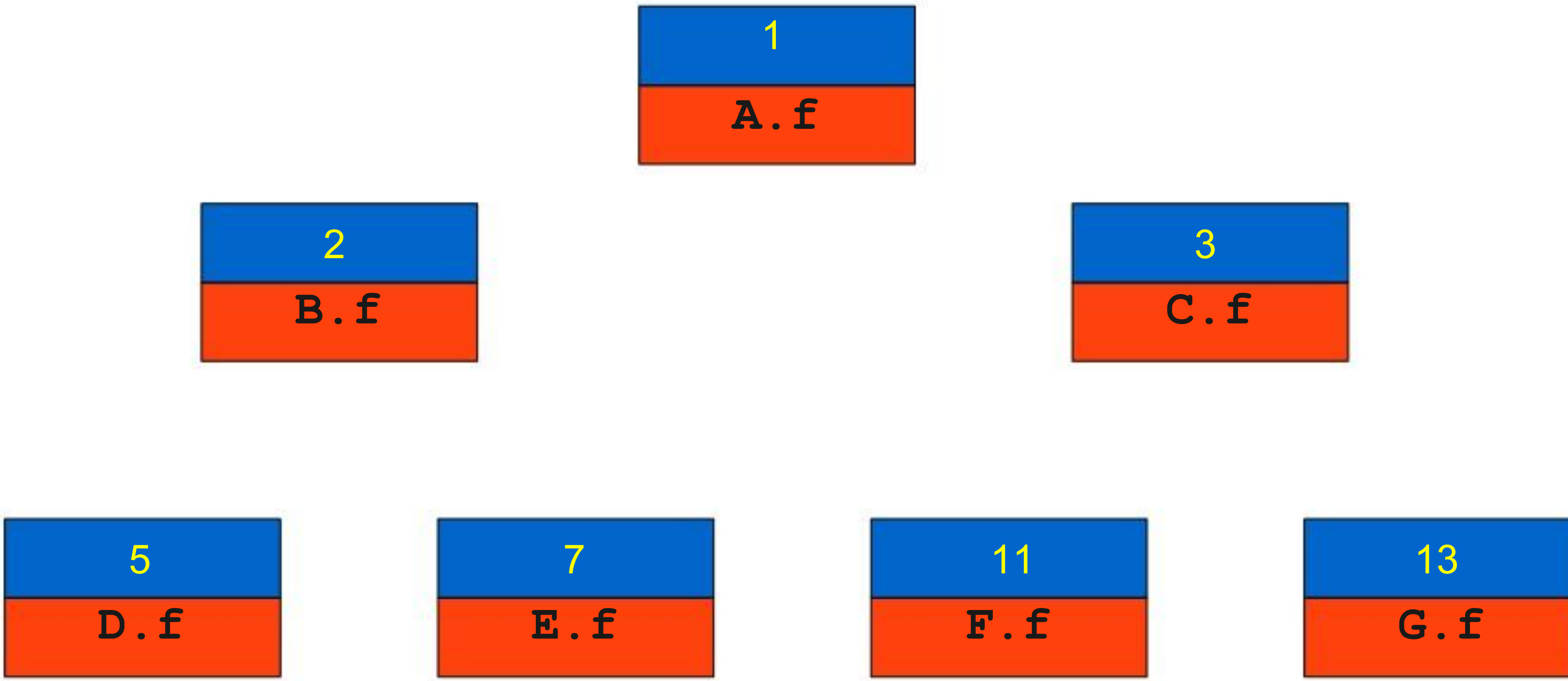
- May want to determine whether the dynamic type is **convertible** to some other type.
- How can we implement this?

Example



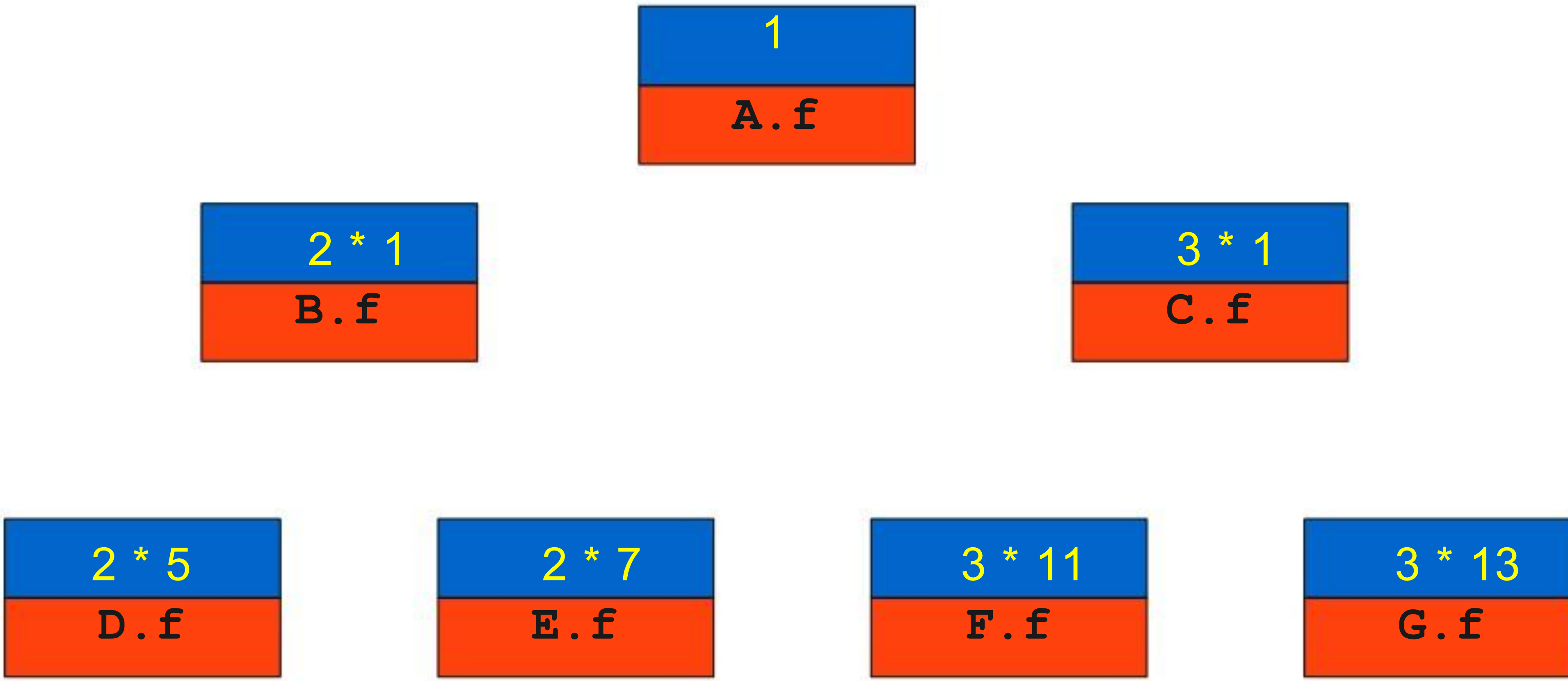
It takes O(log n) time to find out whether X is convertible to Y.
Is there any better idea ?

A Marvelous Idea



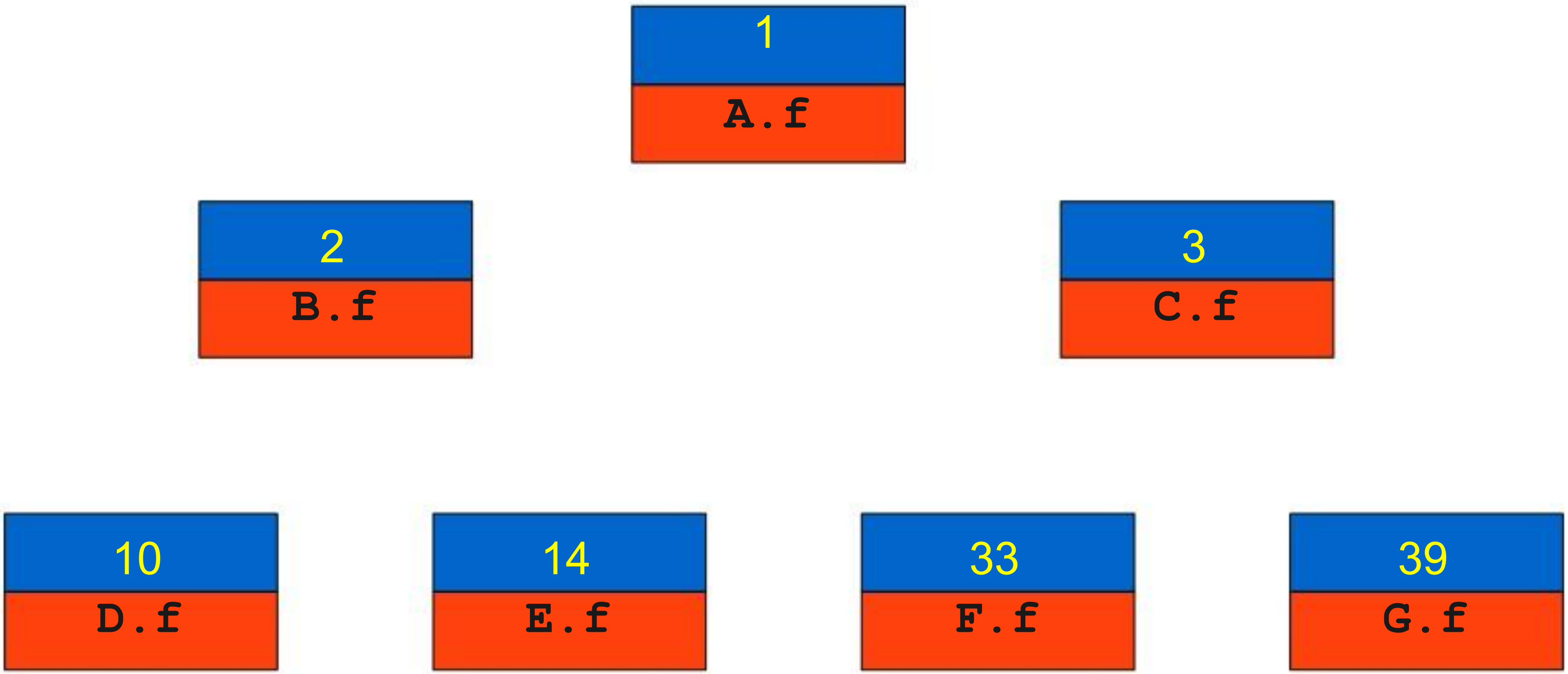
These are prime numbers.

A Marvelous Idea

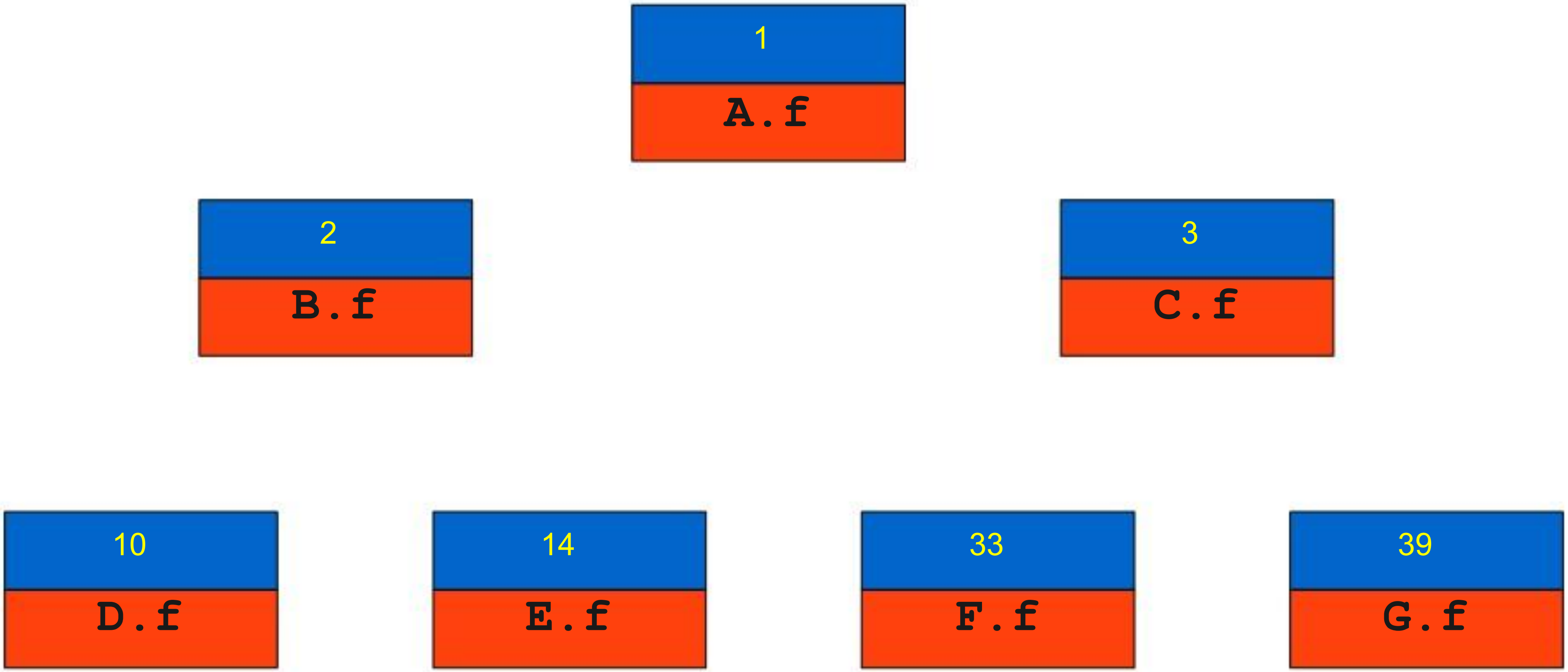


Multiply prime numbers of the class and it’s parent.

A Marvelous Idea

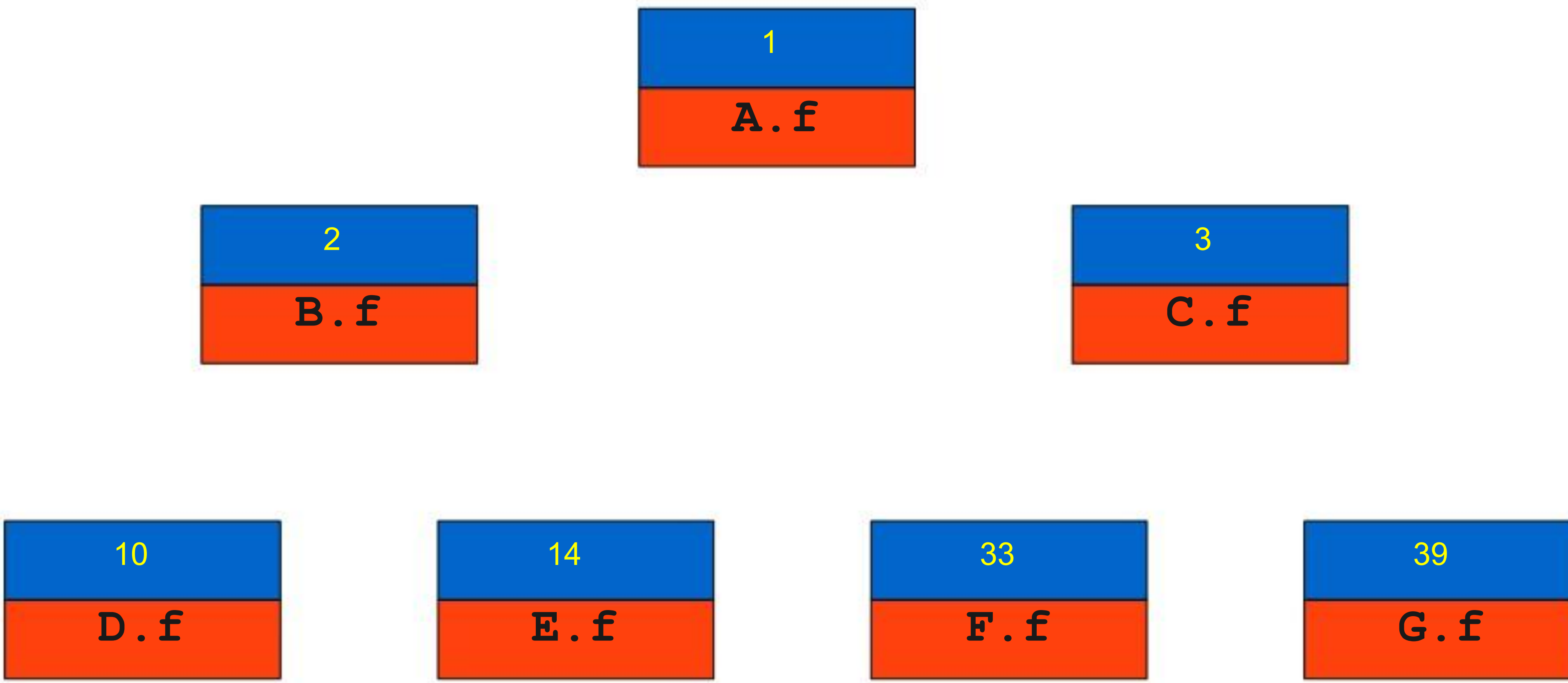


A Marvelous Idea



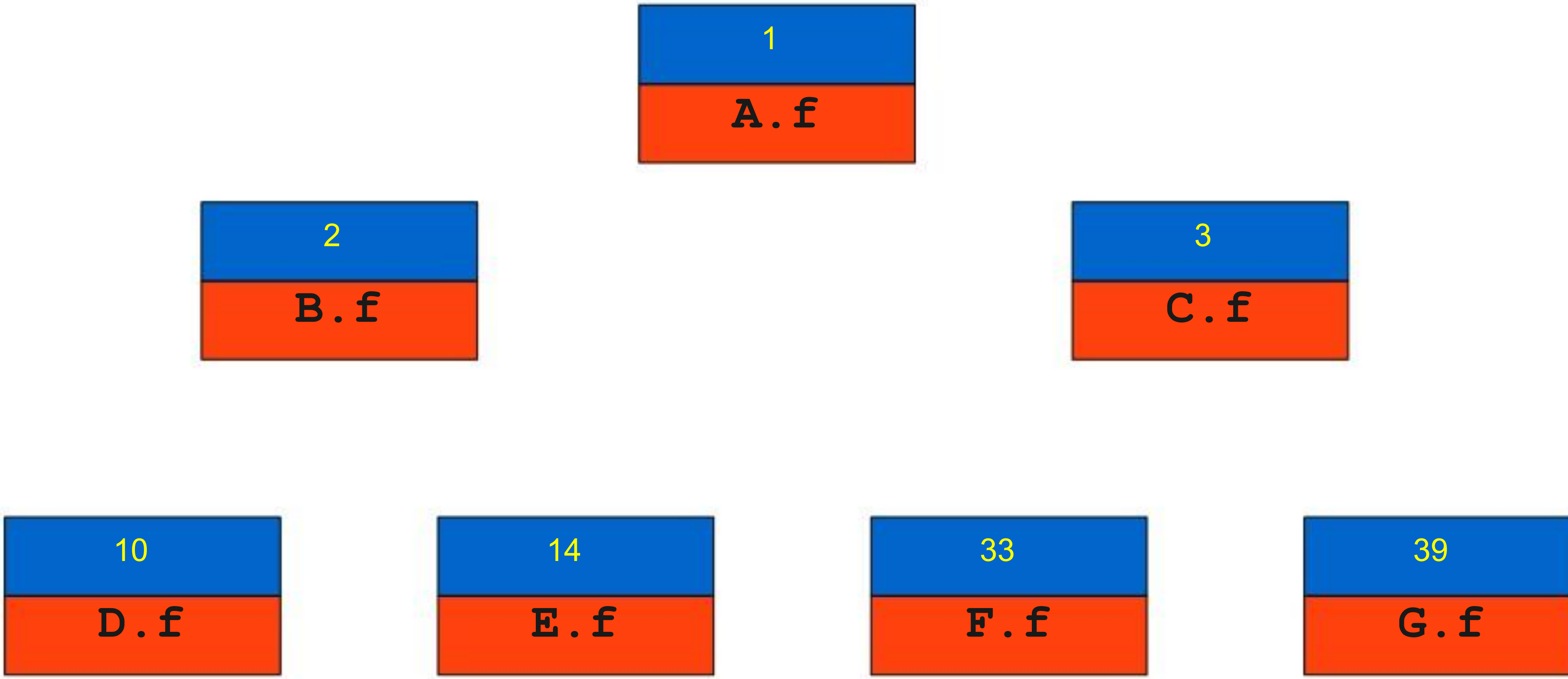

```
G myObject = /* … */
if (myObject instanceof C) {
    /* … */
}
```

A Marvelous Idea




```
G myObject = /* … */
if (myObject->vtable.key % 3 == 0) {
    /* … */
}
```

A Marvelous Idea




```
G myObject = /* … */
if (39 % 3 == 0) {// myObject is an instance of class C
    /* … */
}
```