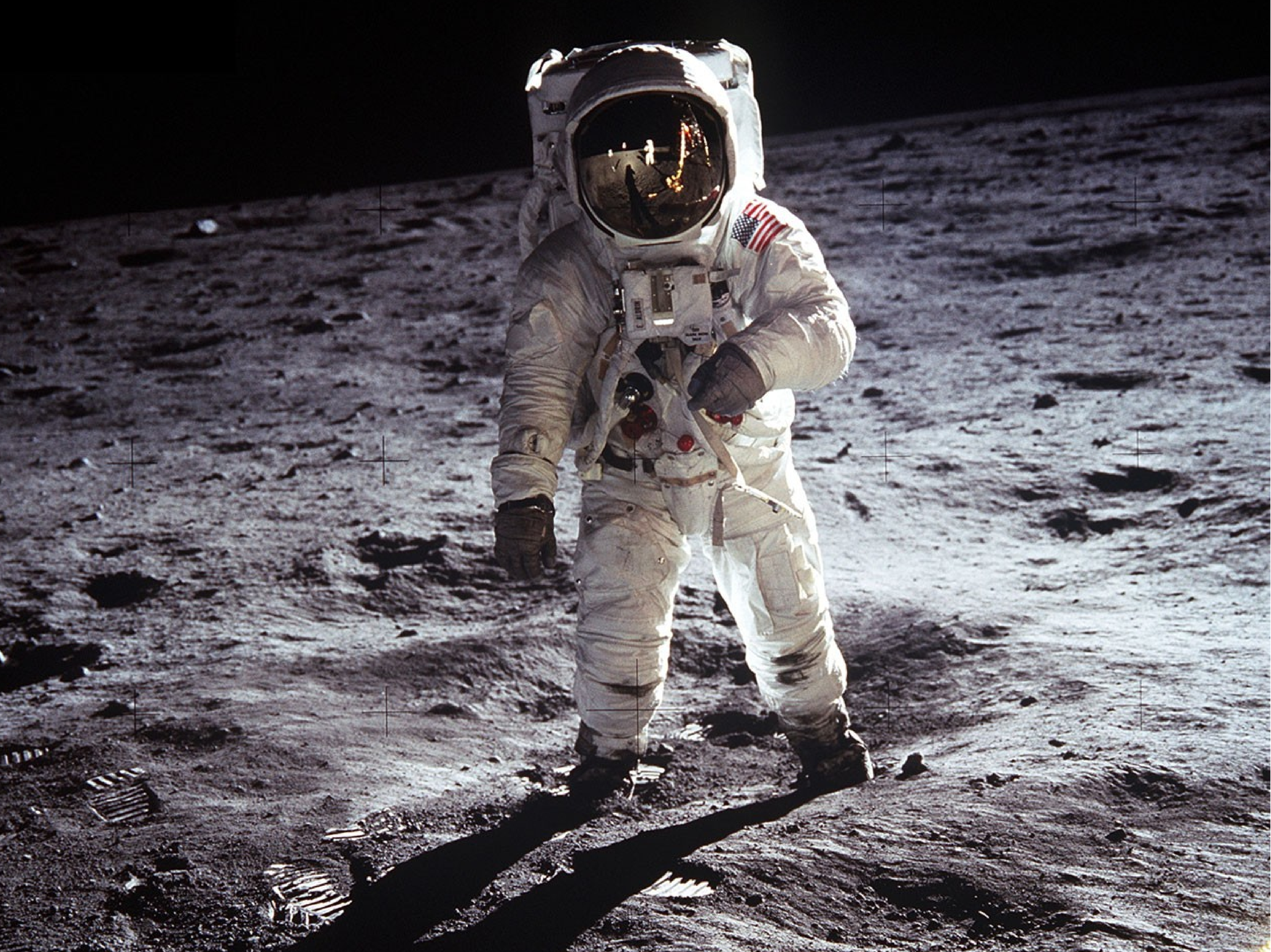
Type-Checking

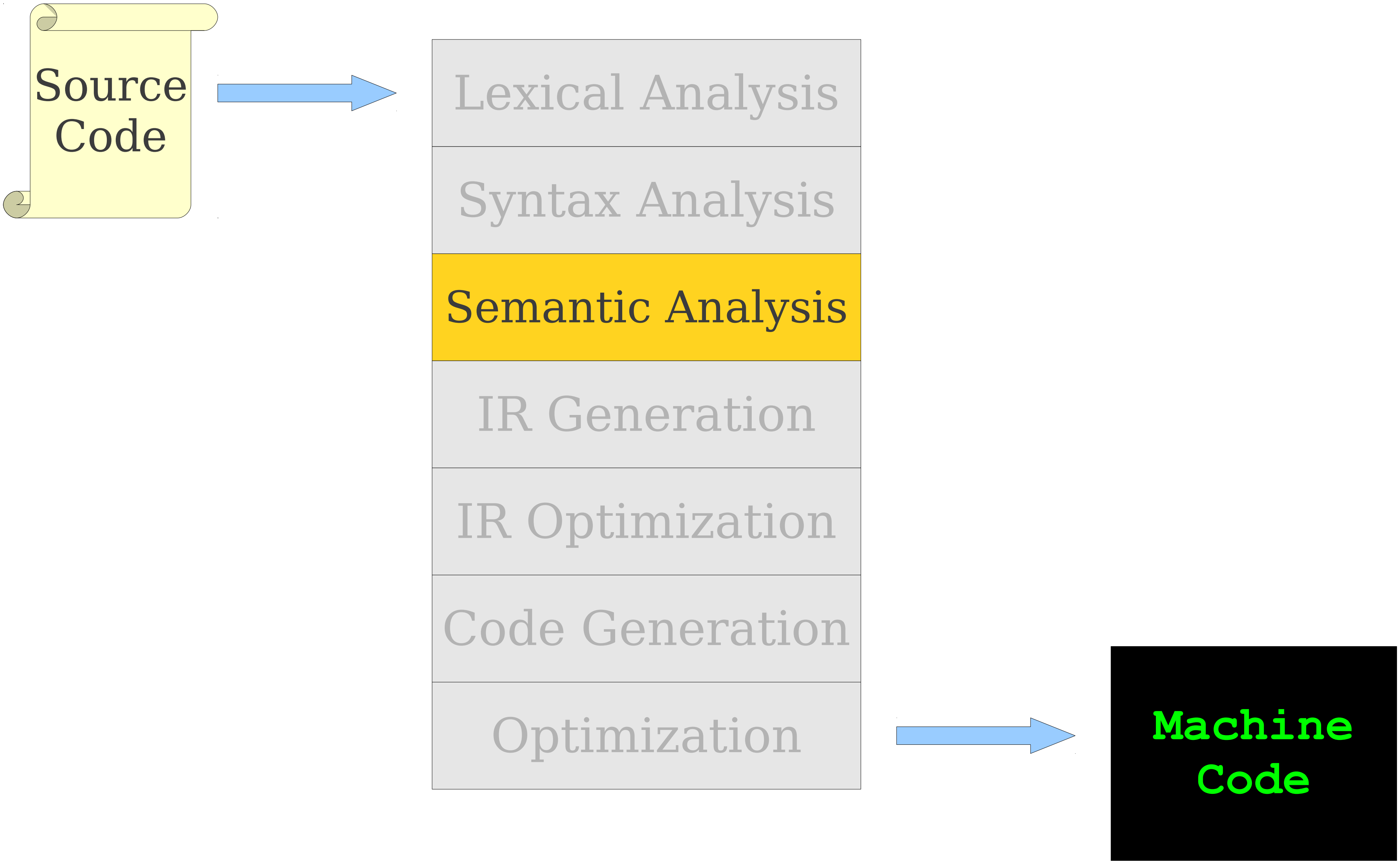
Part II

Announcements

- Programming Assignment 2 due tonight at 11:59PM.
- Programming Assignment 3 (semantic analysis) out:
  - Checkpoint due Monday, July 30 at 11:59PM. **No late submissions accepted**.
  - Remainder due Monday, August 6 at 11:59PM.
- Midterm next Wednesday, 11:00AM – 1:00PM in Thornton 102 (right here!)



Where We Are


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

Review from Last Time

- Static type checking in Decaf consists of two separate processes:
  - Inferring the type of each expression from the types of its components.
  - Confirming that the types of expressions in certain contexts matches what is expected.
- Logically two steps, but you will probably combine into one pass.

Review from Last Time

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

Review from Last Time

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

Review from Last Time

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

Review from Last Time

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

Well-typed expression with wrong type.

Review from Last Time

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

Review from Last Time

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

Expression with type error

Review from Last Time

We write

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

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

Review from Last Time

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \\ S \vdash e_i : T_i \;\text{ for } 1 \leq i \leq n \end{array}}{S \vdash f(e_1, \ldots, e_n) \;:\; U}$$

Review from Last Time

Read rules like this

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : T_i \text{ for } 1 \le i \le n \end{array}}{S \vdash f(e_1, \ldots, e_n) : U}$$

Review from Last Time

- We say that $A \leq B$ if A is convertible to B.
- The **least upper bound** of A and B is the class C where
  - $A \leq C$
  - $B \leq C$
  - $C \leq C'$ for all other upper bounds.
- The least upper bound is denoted $A \vee B$ when it exists.
- A **minimal upper bound** of A and B is
  - an upper bound of A and B
  - that is not larger than any other upper bound.

Review from Last Time

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \\ S \vdash e_i : R_i \ \text{ for } 1 \leq i \leq n \\ R_i \leq T_i \ \text{ for } 1 \leq i \leq n \end{array}}{S \vdash f(e_1, \ldots, e_n) \ : U}$$

Review from Last Time

$$\frac{}{S \vdash \texttt{null} : \texttt{null}\ \text{type}}$$

Overview for Today

- Type-checking **statements**.
- Practical type-checking considerations.
- Type-checking practical language constructs:
  - Function overloading.
  - Specializing overrides.

Using our Type Proofs

- We can now prove the types of various expressions.
- How do we check…
  - … that **`if`** statements have well-formed conditional expressions?
  - … that **`return`** statements actually return the right type of value?
- Use another proof system!

Proofs of Structural Soundness

- Idea: extend our proof system to statements to confirm that they are well-formed.
- We say that

$$S \vdash \mathrm{WF}(\mathit{stmt})$$

  if the statement *stmt* is **well-formed** in scope S.
- The type system is satisfied if for every function *f* with body B in scope S, we can show $S \vdash \mathrm{WF}(B)$.

A Simple Well-Formedness Rule

$$\frac{S \vdash \mathit{expr} : T}{S \vdash \mathrm{WF}(\mathit{expr};)}$$

A Simple Well-Formedness Rule

$$\frac{\boxed{S \vdash \mathit{expr} : T}}{S \vdash \mathrm{WF}(\mathit{expr};)}$$

If we can assign a valid type to an expression in scope S…

A Simple Well-Formedness Rule

$$\frac{S \vdash expr : T}{S \vdash \mathrm{WF}(expr;)}$$

If we can assign a valid type to an expression in scope S…

… then it is a valid statement in scope S.

A More Complex Rule

A More Complex Rule

$$\frac{S \vdash \mathrm{WF}(\mathit{stmt}_1) \quad S \vdash \mathrm{WF}(\mathit{stmt}_2)}{S \vdash \mathrm{WF}(\mathit{stmt}_1\ \mathit{stmt}_2)}$$

Rules for `break`

Rules for `break`

$$\frac{\text{S is in a } \texttt{for} \text{ or } \texttt{while} \text{ loop.}}{S \vdash \mathrm{WF}(\texttt{break;})}$$

A Rule for Loops

A Rule for Loops

$$\frac{\begin{array}{c} S \vdash expr : \texttt{bool} \\ \text{S' is the scope inside the loop.} \\ S' \vdash WF(stmt) \end{array}}{S \vdash WF(\texttt{while (}expr\texttt{)}\ stmt)}$$

Rules for Block Statements

Rules for Block Statements

$$\frac{\text{S' is the scope formed by adding } \mathit{decls} \text{ to S} \qquad \text{S'} \vdash \text{WF}(\mathit{stmt})}{\text{S} \vdash \text{WF}(\texttt{\{}\ \mathit{decls}\ \mathit{stmt}\ \texttt{\}})}$$

Rules for `return`

Rules for `return`

$$\frac{\text{S is in a function returning T} \quad S \vdash expr : T' \quad T' \leq T}{S \vdash \text{WF}(\texttt{return } expr\texttt{;})}$$

$$\frac{\text{S is in a function returning } \texttt{void}}{S \vdash \text{WF}(\texttt{return;})}$$

Checking Well-Formedness

- Recursively walk the AST.
- For each statement:
  - Typecheck any subexpressions it contains.
    - Report errors if **no** type can be assigned.
    - Report errors if the **wrong** type is assigned.
  - Typecheck child statements.
  - Check the overall correctness.

Practical Concerns

Something is Very Wrong Here

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

Something is Very Wrong Here

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

Something is Very Wrong Here

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

Something is Very Wrong Here

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

Facts

Something is Very Wrong Here

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

Facts

Something is Very Wrong Here

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

Facts

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

Something is Very Wrong Here

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

Facts
S ⊢ **x : int**

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

Something is Very Wrong Here

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

Facts
S ⊢ **x : int**

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

Something is Very Wrong Here

```
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 ⊢ **`x`** : **`int`**
S ⊢ **`y`** : **`int`**
S ⊢ **`z`** : **`int`**

Something is Very Wrong Here

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

Facts
S ⊢ **x : int**
S ⊢ **y : int**
S ⊢ **z : int**

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

Something is Very Wrong Here

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

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

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

Something is Very Wrong Here

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

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

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

Something is Very Wrong Here

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

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

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

Something is Very Wrong Here

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

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

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

Something is Very Wrong Here

```
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**

Something is Very Wrong Here

```
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**

Something is Very Wrong Here

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

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

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

Something is Very Wrong Here

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

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

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**

Something is Very Wrong Here

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

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

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**

Something is Very Wrong Here

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

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

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**

Something is Very Wrong Here

```
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**

Something is Very Wrong Here

```
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**

Something is Very Wrong Here

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

$$\frac{S \vdash e_1 : T_1 \qquad S \vdash e_2 : T_2 \qquad T_1 \le T_2 \text{ or } T_2 \le T_1}{S \vdash e_1 \texttt{ == } e_2 : \texttt{bool}}$$

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**

Something is Very Wrong Here

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

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

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**
$S \vdash$ **x == z** : **bool**

Something is Very Wrong Here

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

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

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**

Something is Very Wrong Here

```
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`
$S \vdash$ `x == z` : `bool`

Something is Very Wrong Here

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

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

```
>
```

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

Something is Very Wrong Here

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

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

```
> 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`

Something is Very Wrong Here

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

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

```
> 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`

Something is Very Wrong Here

```
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
```

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`

Something is Very Wrong Here

```
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`

Something is Very Wrong Here

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

Something is Very Wrong Here

```
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`

Something is Very Wrong Here

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

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

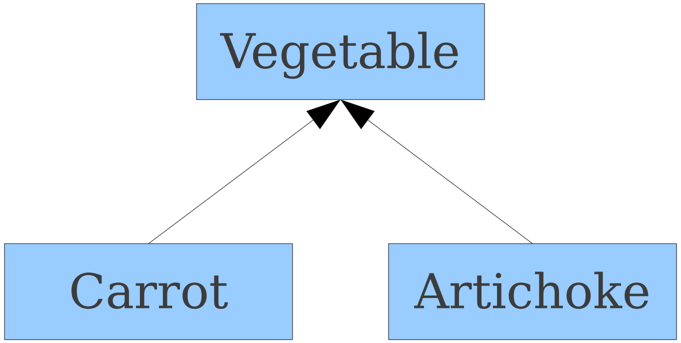
```
> Error: Cannot compare int and bool
  Error: Cannot compare ??? and bool
  Error: Cannot compare ??? and bool
```

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

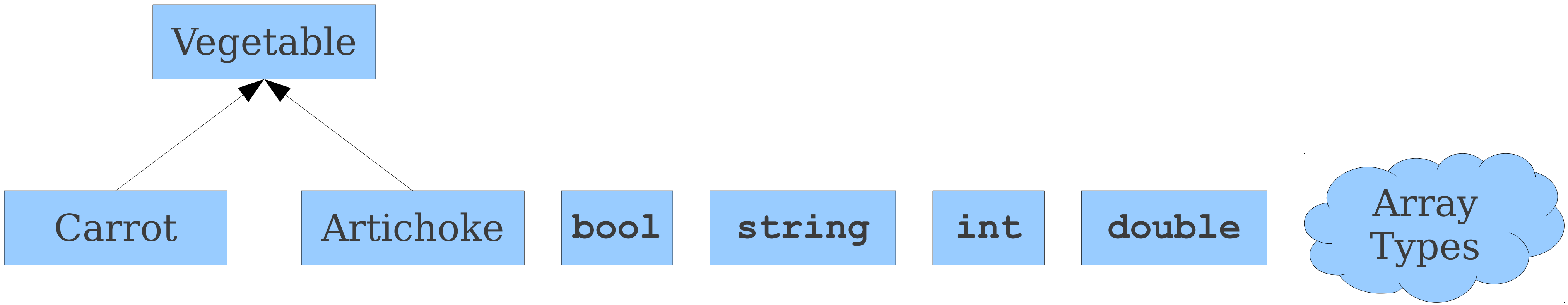
Cascading Errors

- A **static type error** occurs when we cannot prove that an expression has a given type.
- Type errors can easily cascade:
  - Can't prove a type for $\mathbf{e_1}$, so can't prove a type for $\mathbf{e_1 + e_2}$, so can't prove a type for $\mathbf{(e_1 + e_2) + e_3}$, etc.
- How do we resolve this?

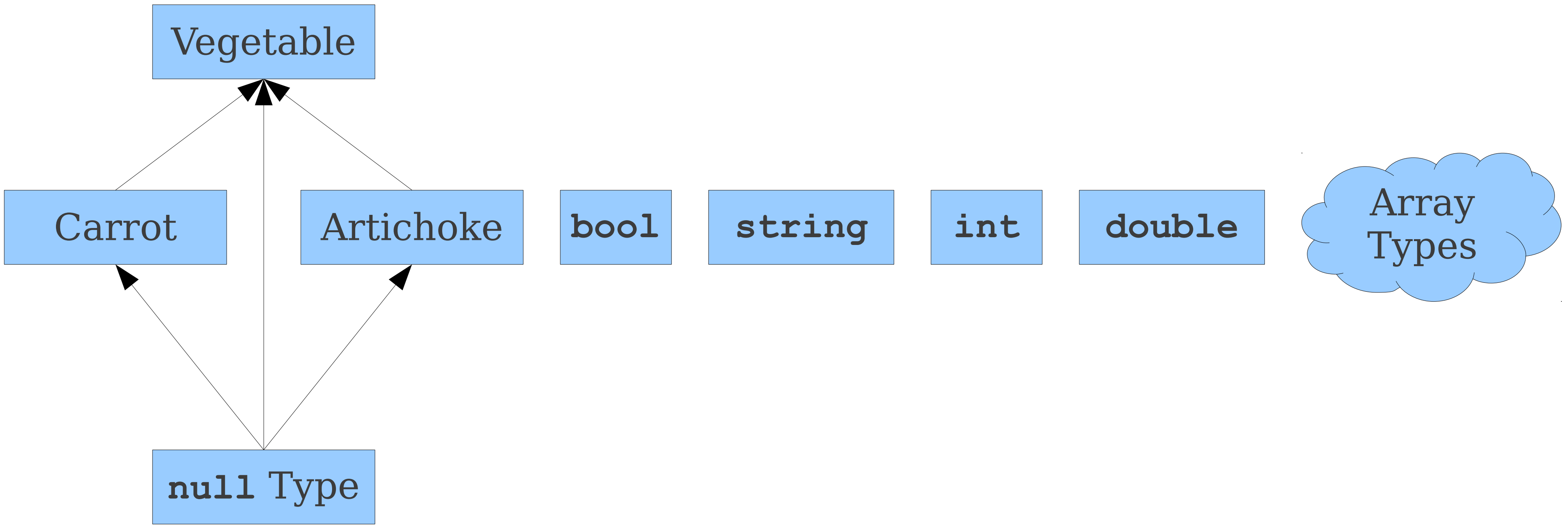
The Shape of Types



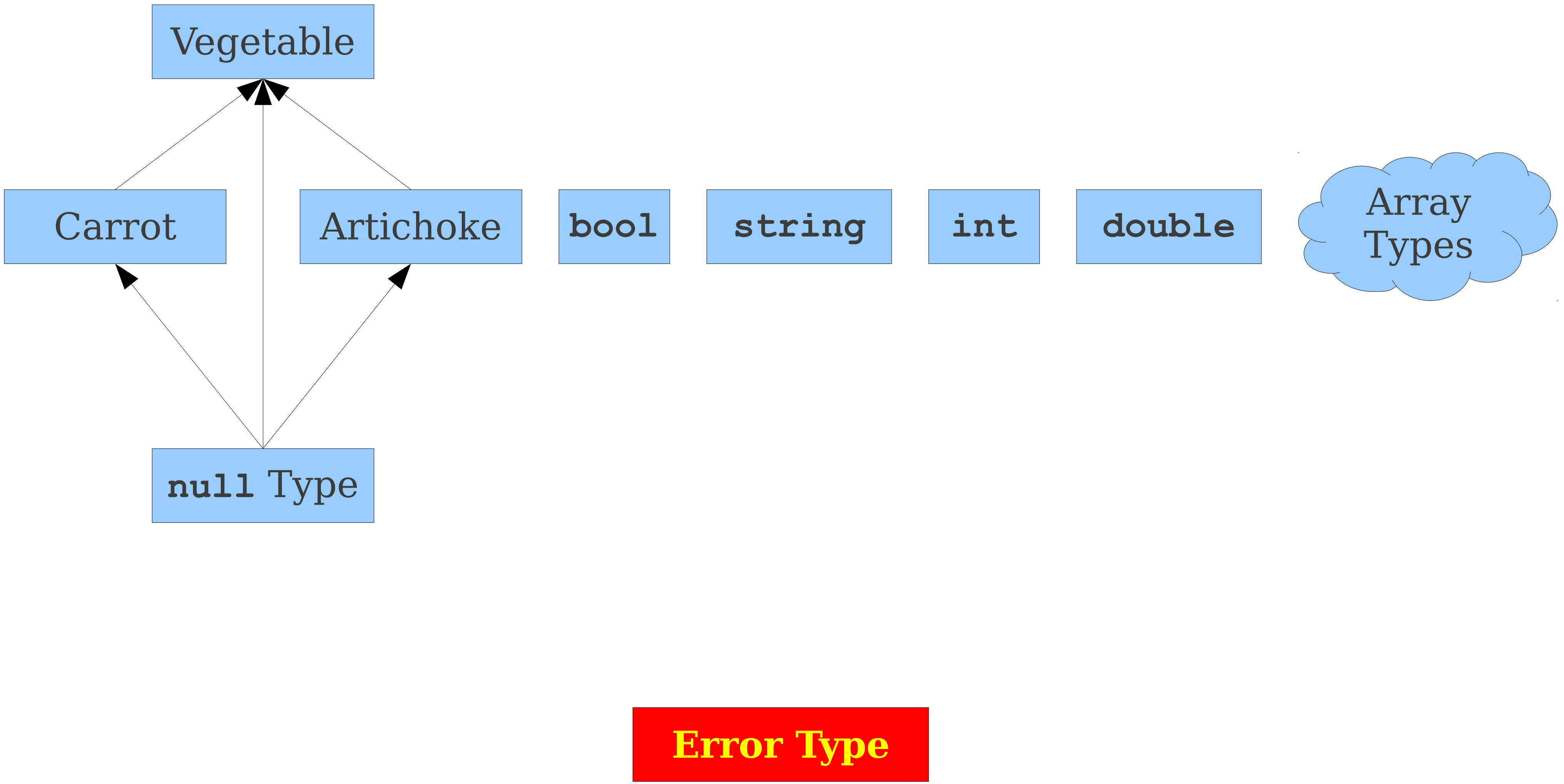
The Shape of Types



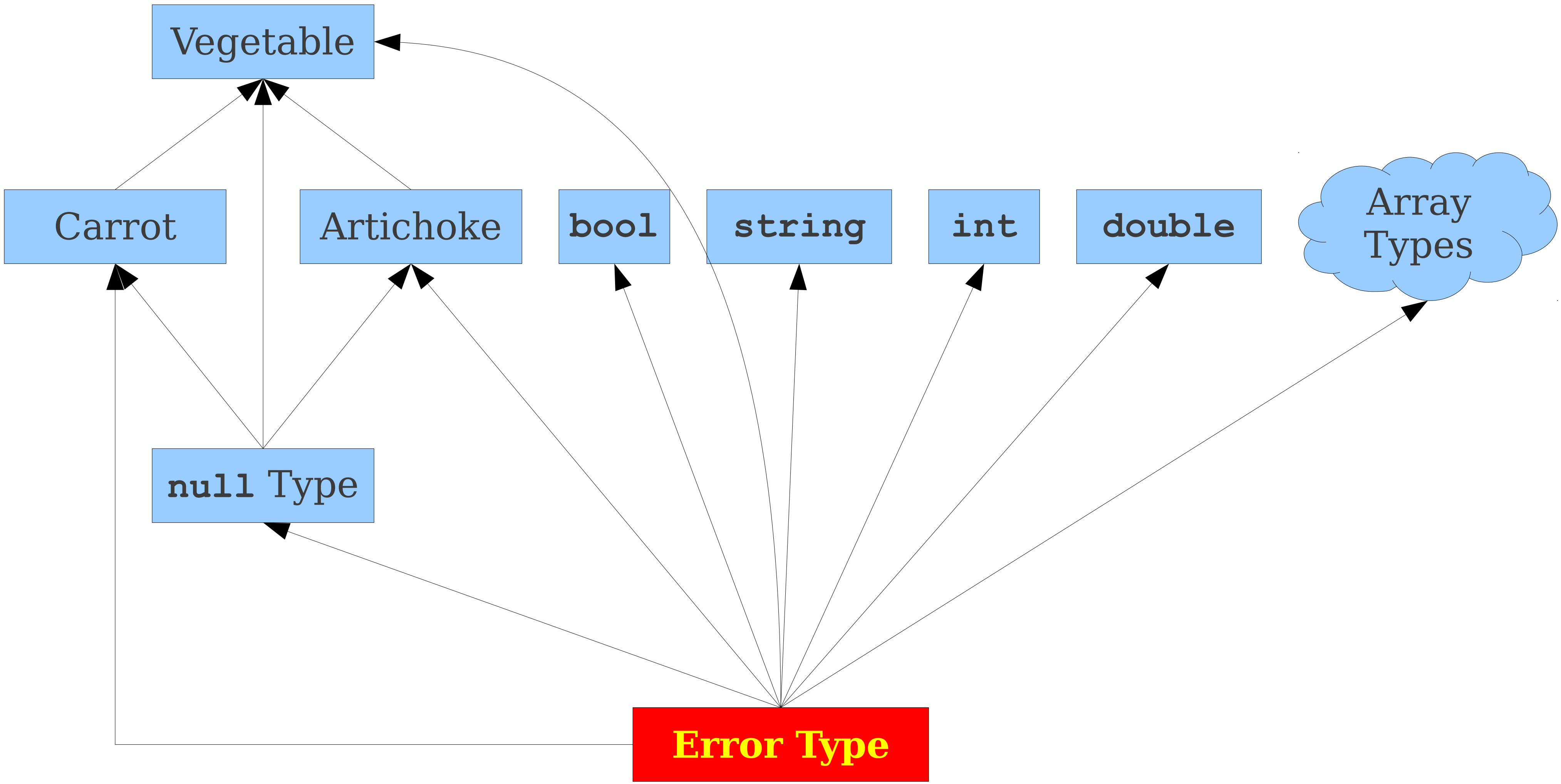
The Shape of Types



The Shape of Types



The Shape of Types



The Error Type

- Introduce a new type representing an error into the type system.
- The **error type** is less than all other types and is denoted **$\perp$**.
  - It is sometimes called the **bottom type**.
- By definition, $\perp \leq A$ for any type A.
- On discovery of a type error, pretend that we can prove the expression has type $\perp$.
- Update our inference rules to support $\perp$.

Updated Rules for Addition

$$\frac{\begin{array}{l} S \vdash e_1 : \mathbf{double} \\ S \vdash e_2 : \mathbf{double} \end{array}}{S \vdash e_1 + e_2 : \mathbf{double}}$$

Updated Rules for Addition

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \end{array}}{S \vdash e_1 + e_2 : \texttt{\textbf{double}}}$$

Updated Rules for Addition

$$\frac{\begin{array}{c} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq \mathbf{double} \\ T_2 \leq \mathbf{double} \end{array}}{S \vdash e_1 + e_2 : \mathbf{double}}$$

Updated Rules for Addition

$$\frac{\begin{array}{c} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq \textbf{double} \\ T_2 \leq \textbf{double} \end{array}}{S \vdash e_1 + e_2 : \textbf{double}}$$

What does this mean?

Updated Rules for Addition

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \leq \texttt{double} \quad T_2 \leq \texttt{double}}{S \vdash e_1 + e_2 : \texttt{double}}$$

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \leq \texttt{int} \quad T_2 \leq \texttt{int}}{S \vdash e_1 + e_2 : \texttt{int}}$$

Updated Rules for Addition

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad \boxed{T_1 \leq \texttt{double} \quad T_2 \leq \texttt{double}}}{S \vdash e_1 + e_2 : \texttt{double}}$$

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad \boxed{T_1 \leq \texttt{int} \quad T_2 \leq \texttt{int}}}{S \vdash e_1 + e_2 : \texttt{int}}$$

Prevents errors from propagating.

Updated Rules for Addition

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \leq \texttt{double} \quad T_2 \leq \texttt{double}}{S \vdash e_1 + e_2 : \texttt{double}}$$

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \leq \texttt{int} \quad T_2 \leq \texttt{int}}{S \vdash e_1 + e_2 : \texttt{int}}$$

Updated Rules for Addition

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \leq \texttt{double} \quad T_2 \leq \texttt{double}}{S \vdash e_1 + e_2 : \texttt{double}}$$

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \leq \texttt{int} \quad T_2 \leq \texttt{int}}{S \vdash e_1 + e_2 : \texttt{int}}$$

What happens if both operands have error type?

Error-Recovery in Practice

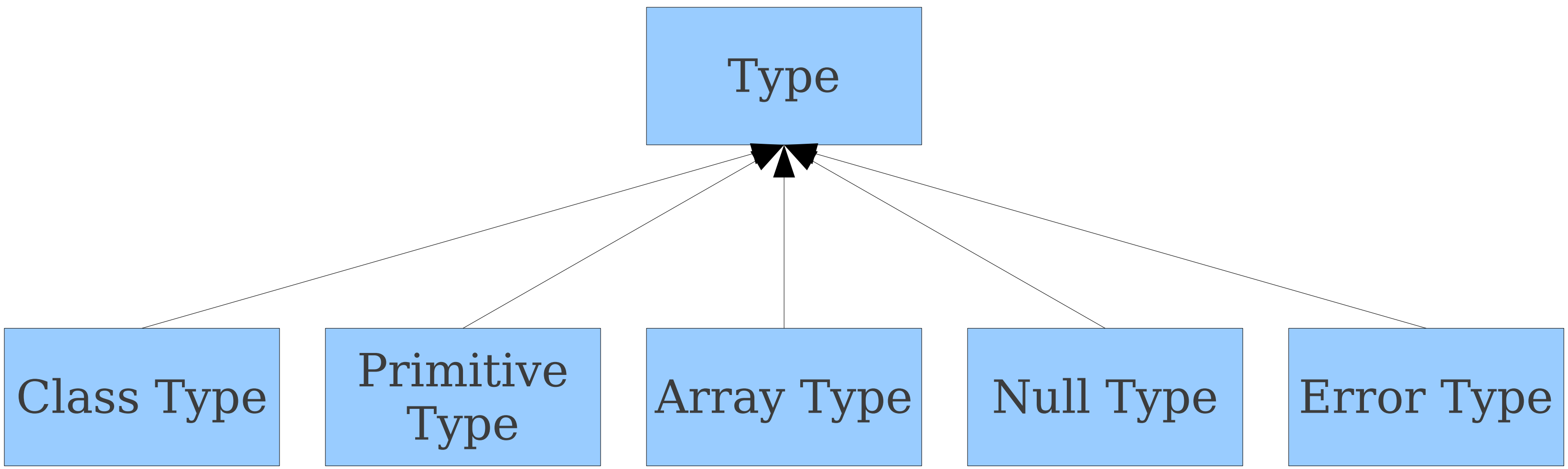
- In your semantic analyzer, you will need to do some sort of error recovery.
- We provide an error type **`Type::errorType`**.
- But what about other cases?
  - Calling a nonexistent function.
  - Declaring a variable of a bad type.
  - Treating a non-array as an array.
- There are no right answers to these questions; just better and worse choices.

Implementing Convertibility

- How do we implement the ≤ operator we've described so far?
- Lots of cases:

From \ To	Class Type	Primitive Type	Array Type	Null Type	Error Type
Class Type	If same or inherits from	No	No	No	No
Primitive Type	No	If same type	No	No	No
Array Type	No	No	If underlying types match	No	No
Null Type	**Yes**	No	No	**Yes**	No
Error Type	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

A Hierarchy for Types



Methods You Might Want...

- `virtual bool Type::IsIdenticalTo(Type* other);`
  - Returns whether two types represent the same actual type.
- `virtual bool Type::IsConvertibleTo(Type* other);`
  - Returns whether one type is convertible to some other type.

Function Overloading

Function Overloading

- Two functions are said to be **overloads** of one another if they have the same name but a different set of arguments.
- At compile-time, determine which function is meant by inspecting the types of the arguments.
- Report an error if no one function is the best function.

Overloading Example

```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

```
Function();
Function(137);
Function(42.0);
Function(new Base);
Function(new Derived);
```

Overloading Example

```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

```
Function();
Function(137);
Function(42.0);
Function(new Base);
Function(new Derived);
```

Implementing Overloading

```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

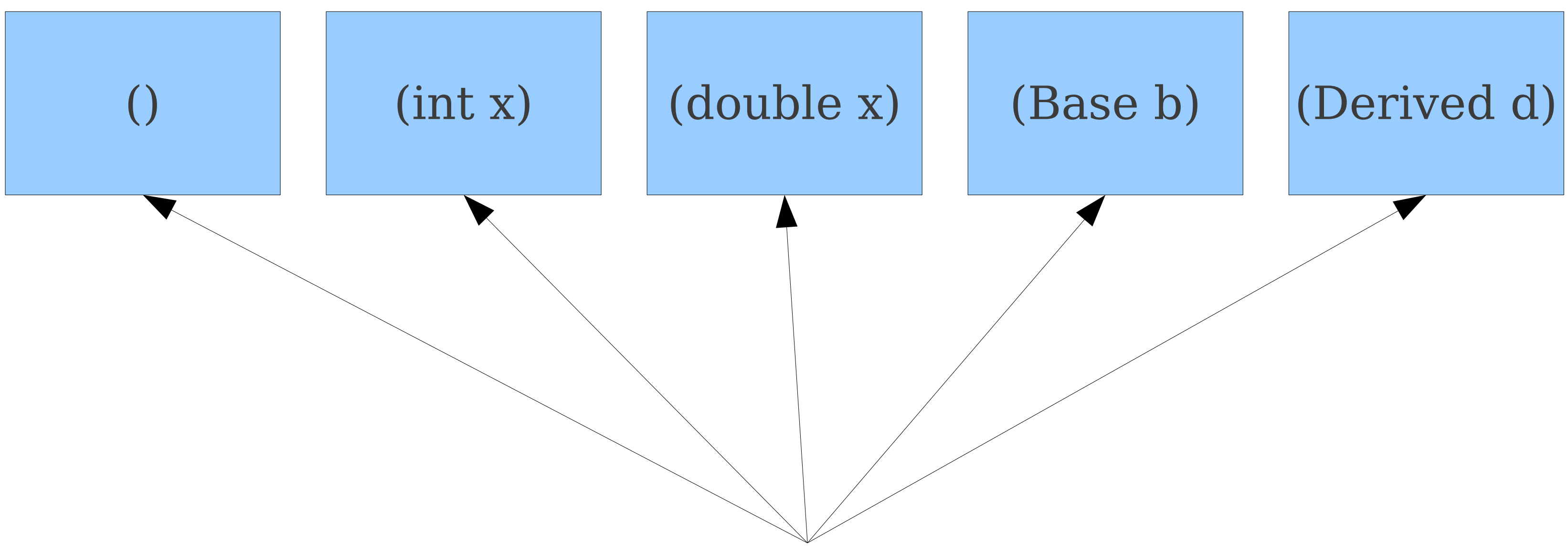
Implementing Overloading

```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

`Function(137);`

Implementing Overloading

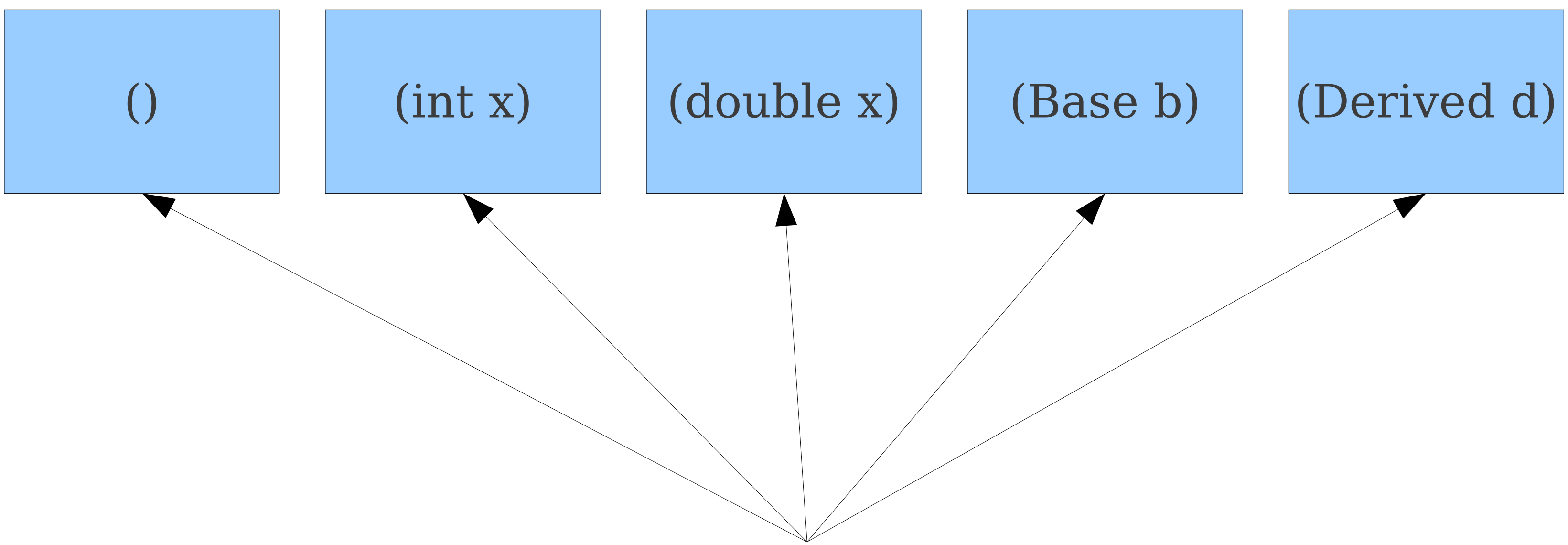
```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```




```
Function(137);
```

Implementing Overloading

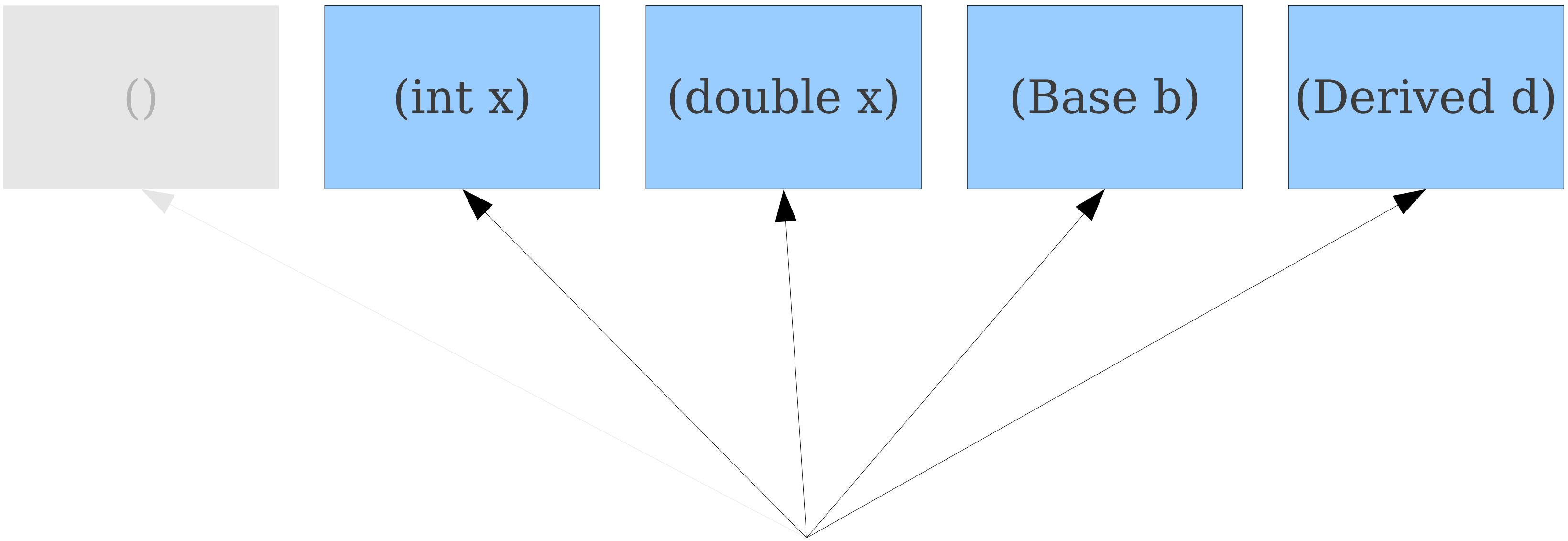
```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```




```
Function(137);
```

Implementing Overloading

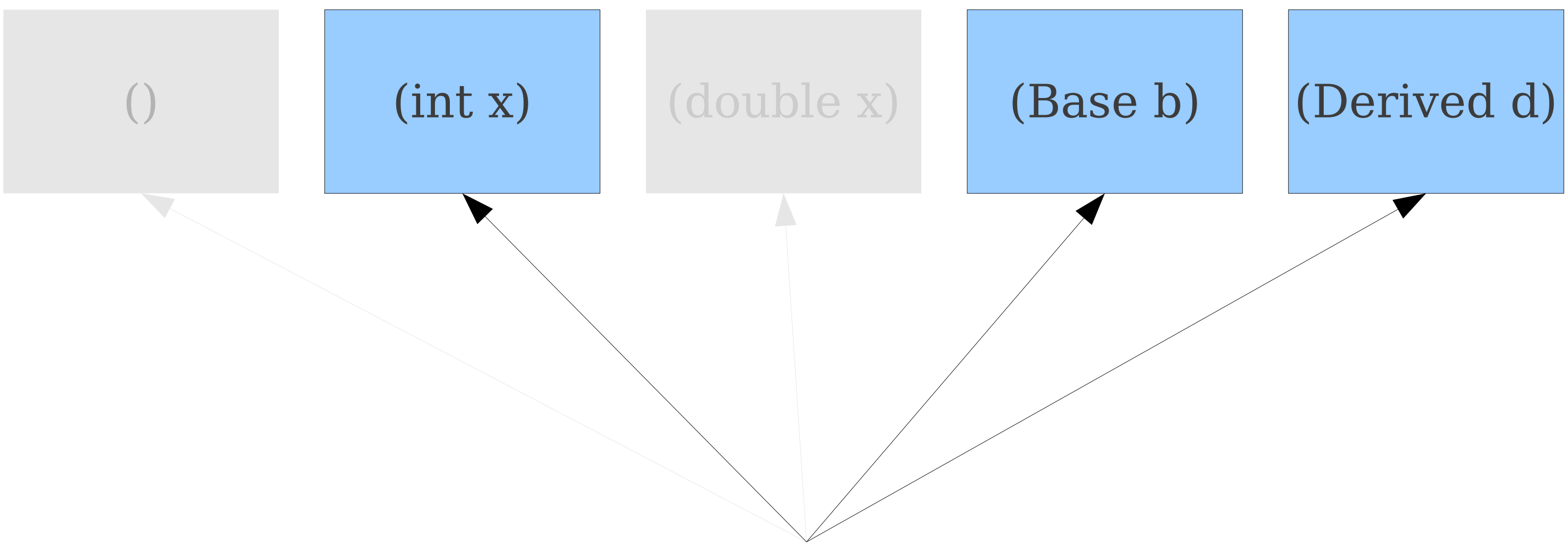
```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```




```
Function(137);
```

Implementing Overloading

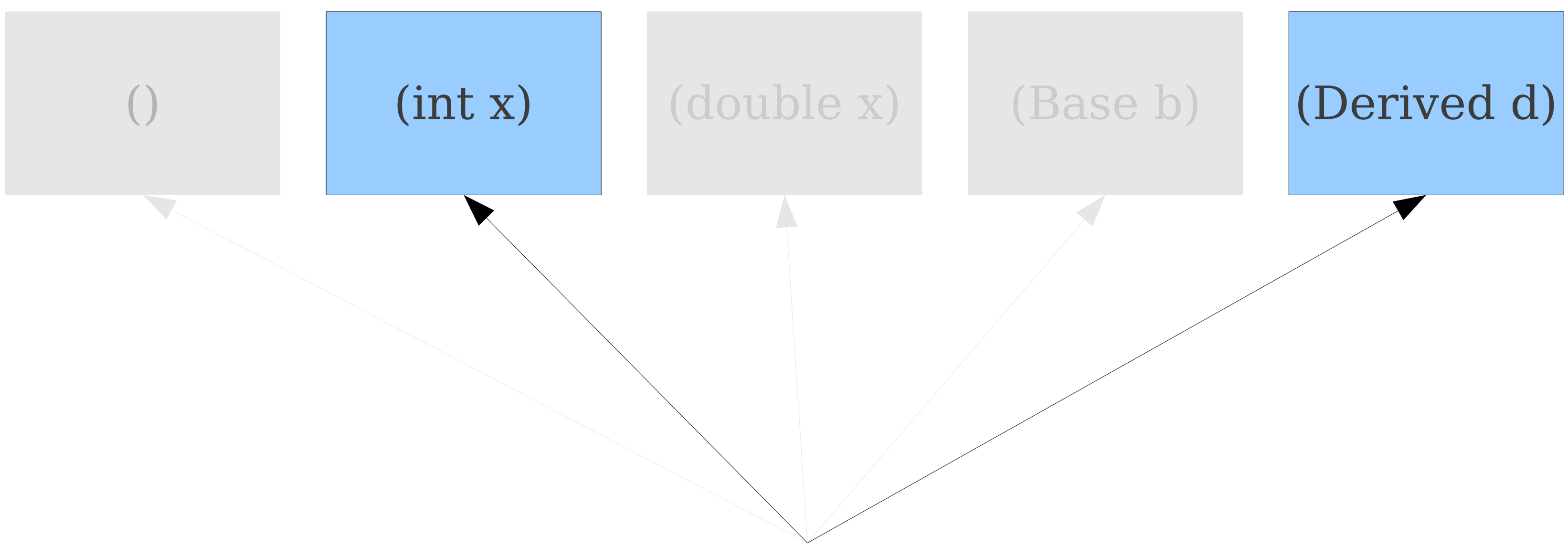
```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```




```
Function(137);
```

Implementing Overloading

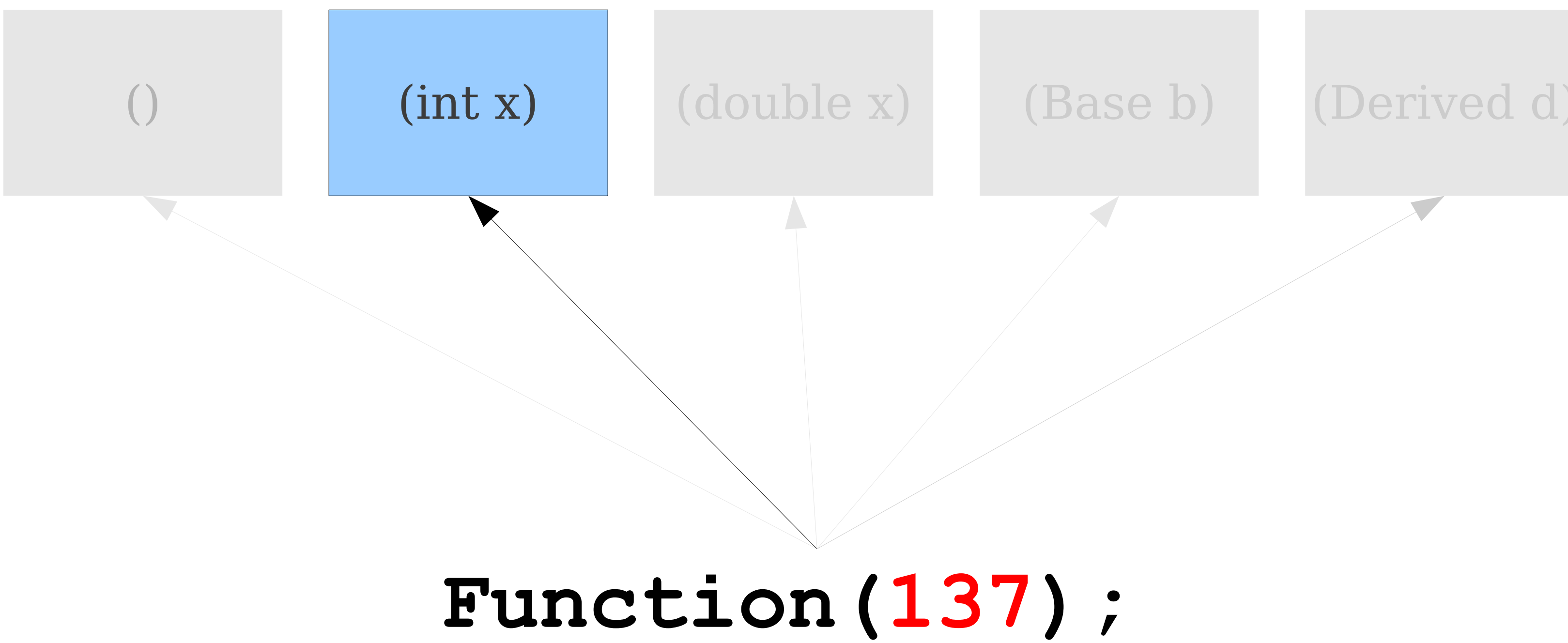
```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```




```
Function(137);
```

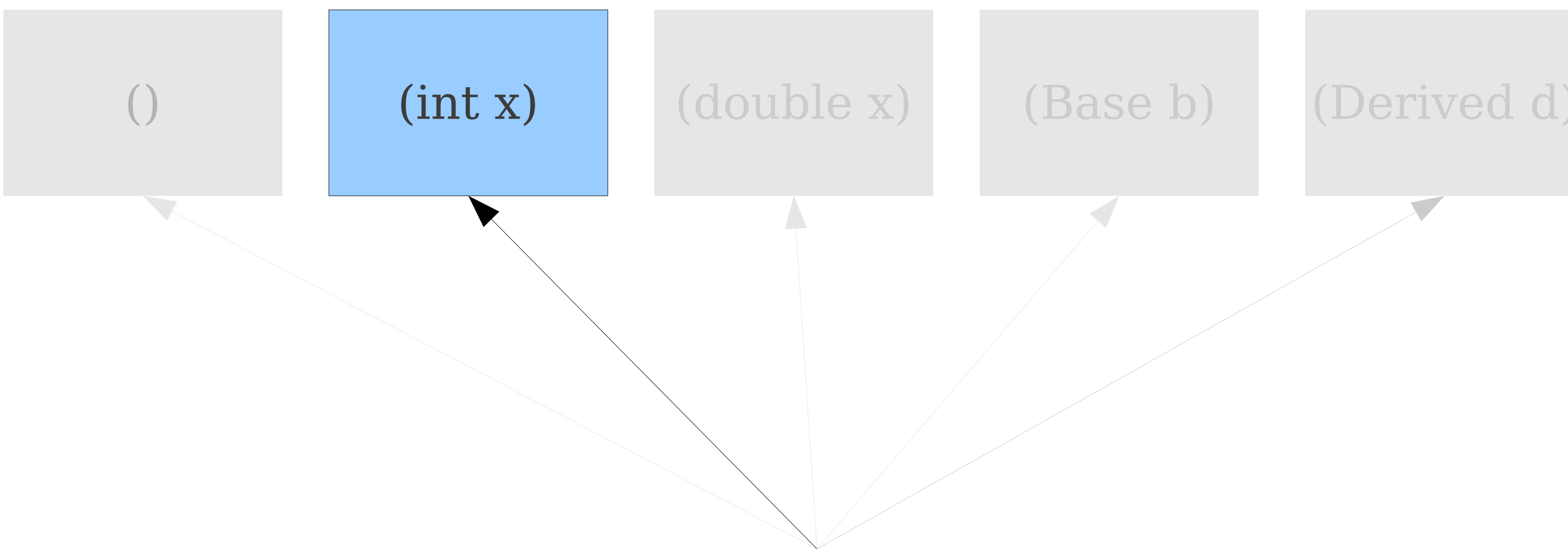
Implementing Overloading

```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```



Implementing Overloading

```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```




```
Function(137);
```

Simple Overloading

- We begin with a set of overloaded functions.
- After filtering out functions that cannot match, we have a **candidate set** (C++ terminology) or set of **potentially applicable methods** (Java-speak).
- If no functions are left, report an error.
- If exactly one function left, choose it.
- (We'll deal with two or more in a second)

Overloading with Inheritance

```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

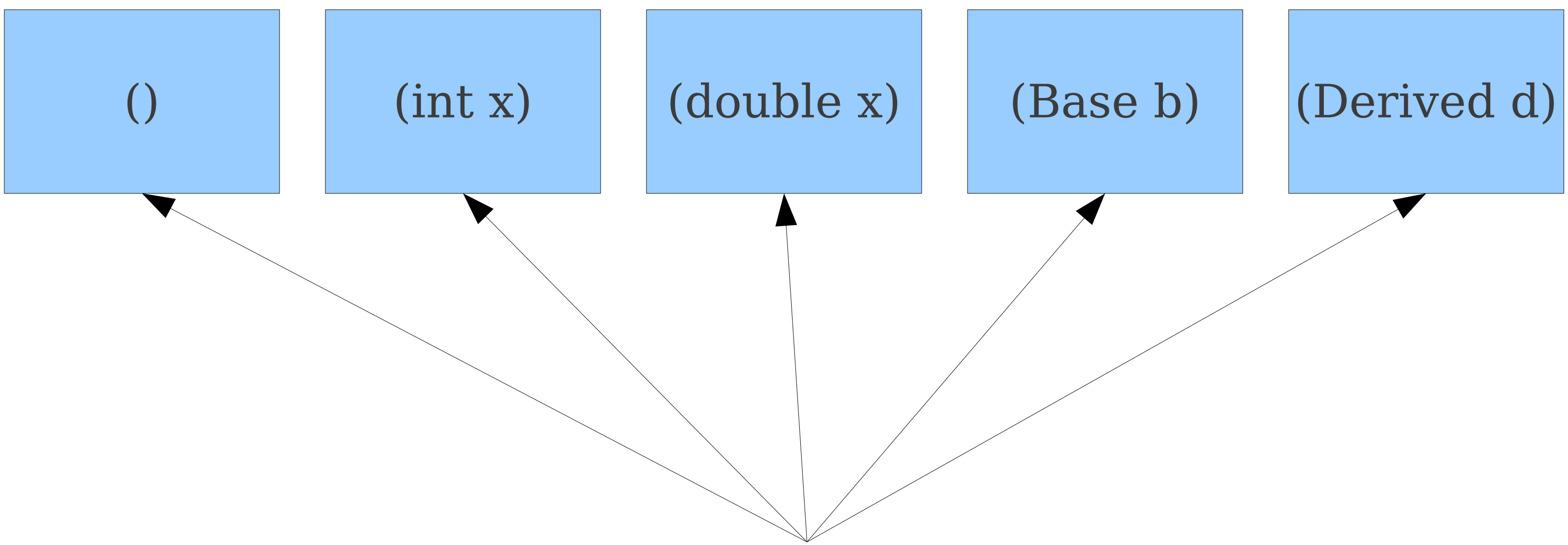
Overloading with Inheritance

```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

```
Function(new Derived);
```

Overloading with Inheritance

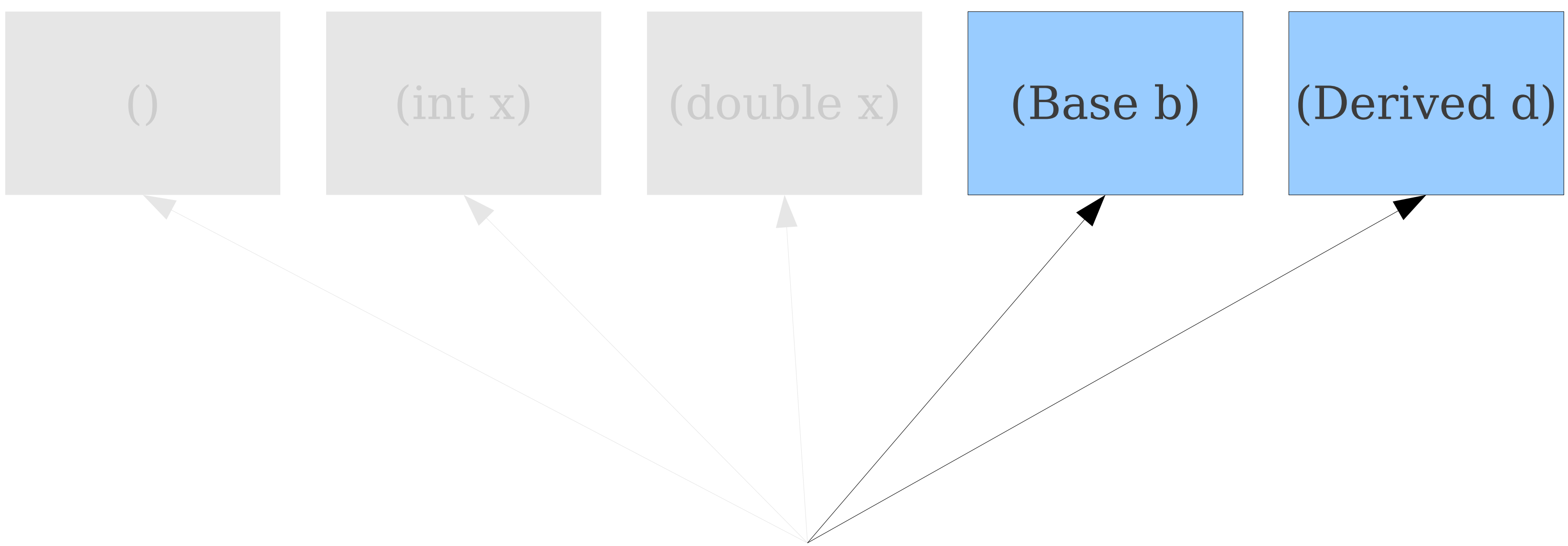
```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```




```
Function(new Derived);
```

Overloading with Inheritance

```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

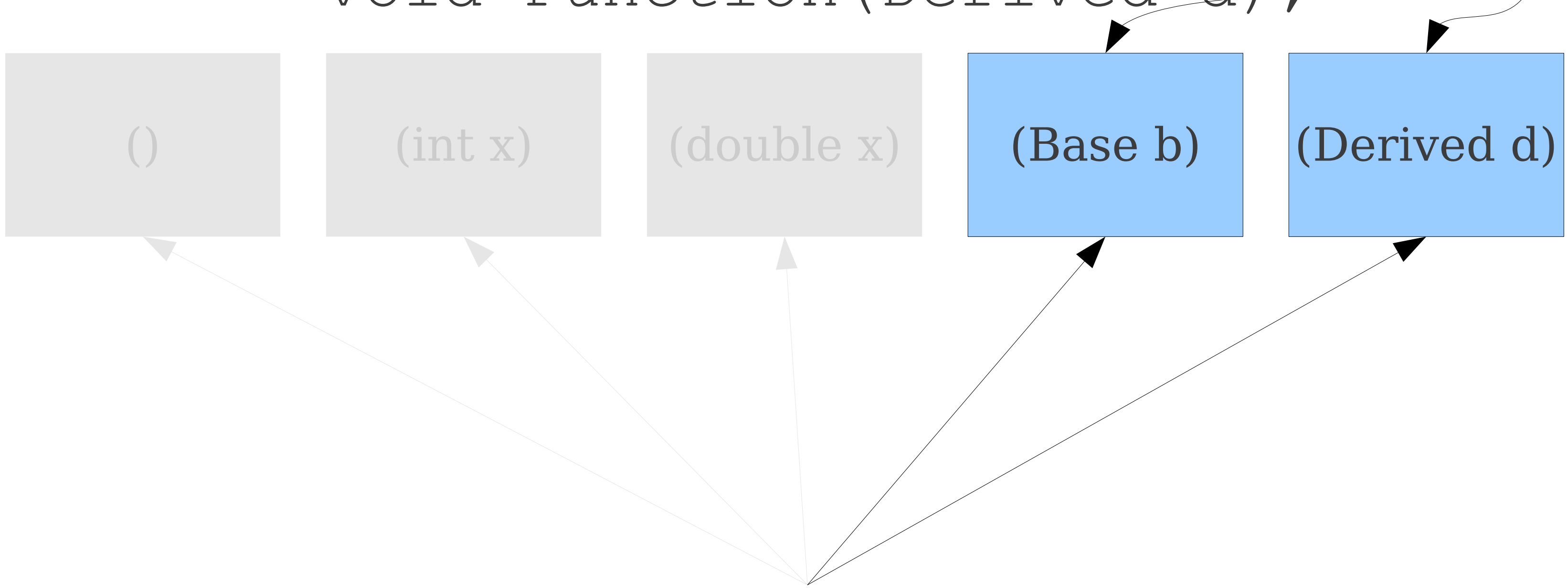



```
Function(new Derived);
```

Overloading with Inheritance

```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```

How do we compare these?



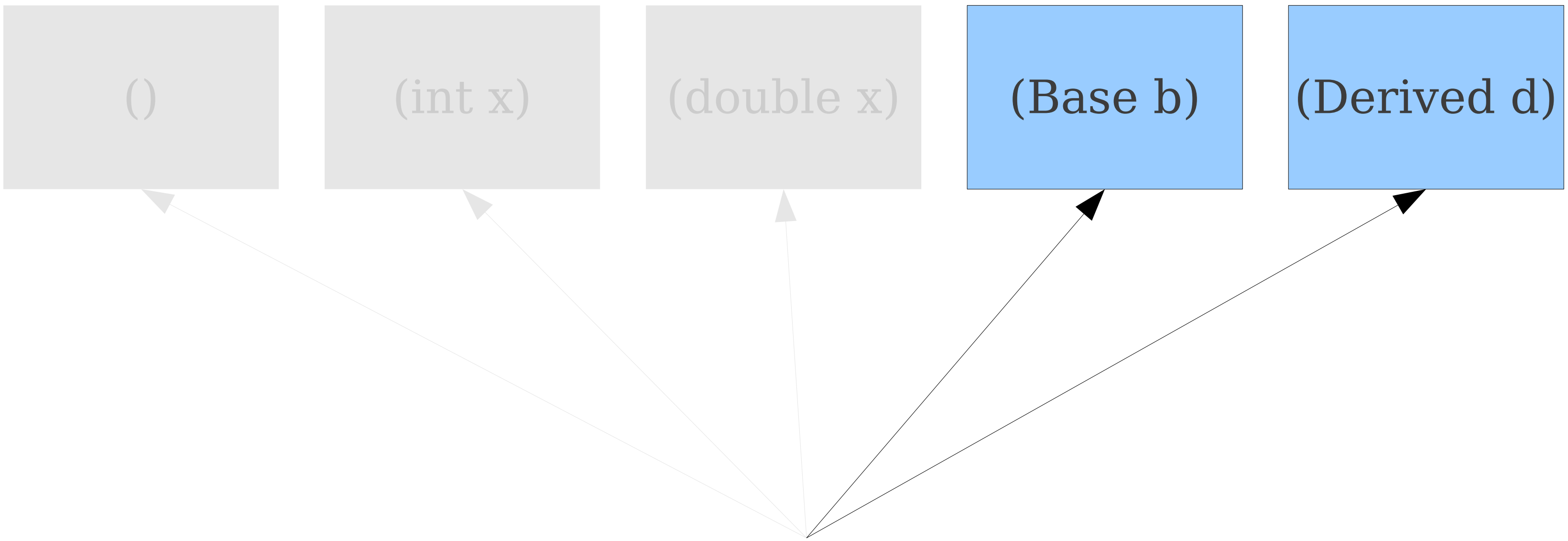

```
Function(new Derived);
```

Finding the Best Match

- Choose one function over another if it's strictly more specific.
- Given two candidate functions A and B with argument types $A_1, A_2, \ldots, A_n$ and $B_1, B_2, \ldots, B_n$, we say that A <: B if $A_i \leq B_i$ for all i, $1 \leq i \leq n$.
  - This relation is also a **partial order**.
- A candidate function A is the **best match** if for any candidate function B, A <: B.
  - It's at least as good any other match.
- If there is a best match, we choose that function. Otherwise, the call is ambiguous.

Overloading with Inheritance

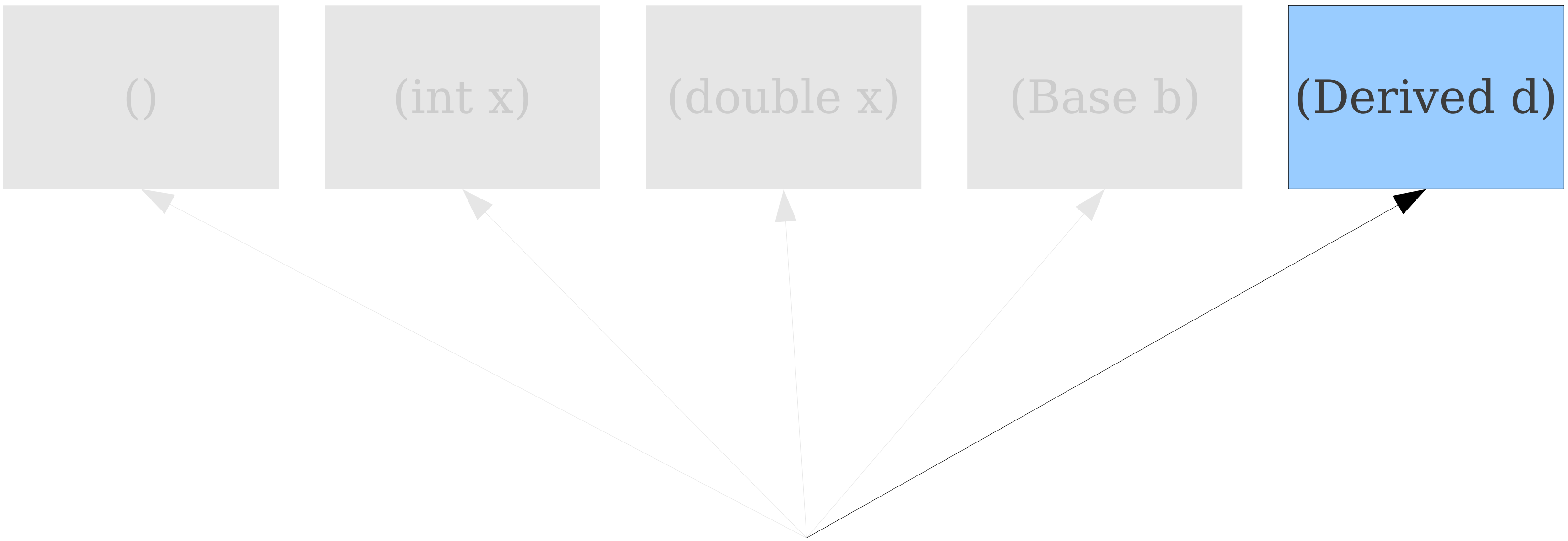
```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```




```
Function(new Derived);
```

Overloading with Inheritance

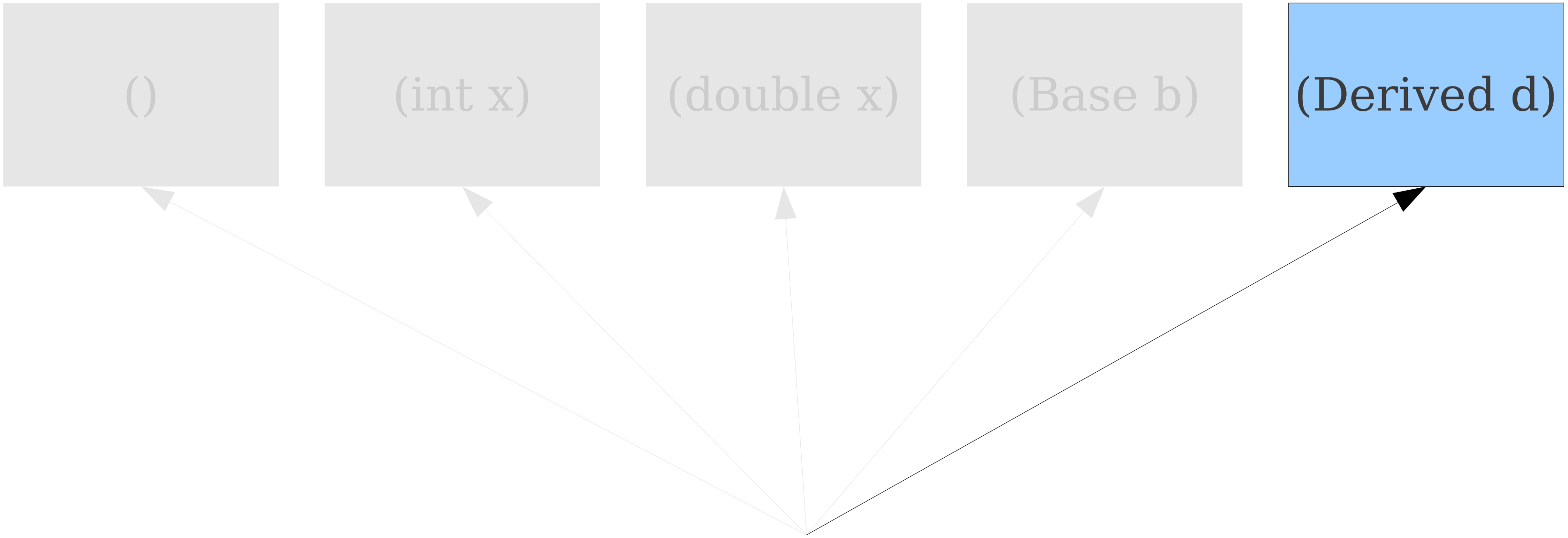
```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```




```
Function(new Derived);
```

Overloading with Inheritance

```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);
```




```
Function(new Derived);
```

Ambiguous Calls

```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Base b1, Derived d2);
```

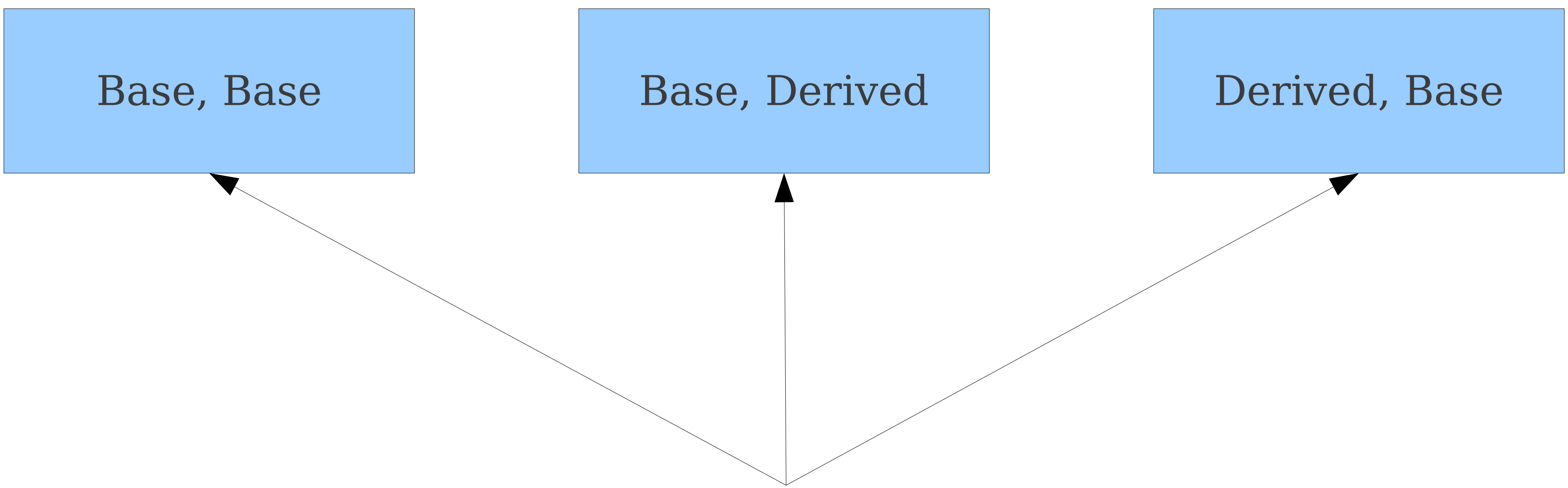
Ambiguous Calls

```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Base b1, Derived d2);
```

```
Function(new Derived, new Derived);
```

Ambiguous Calls

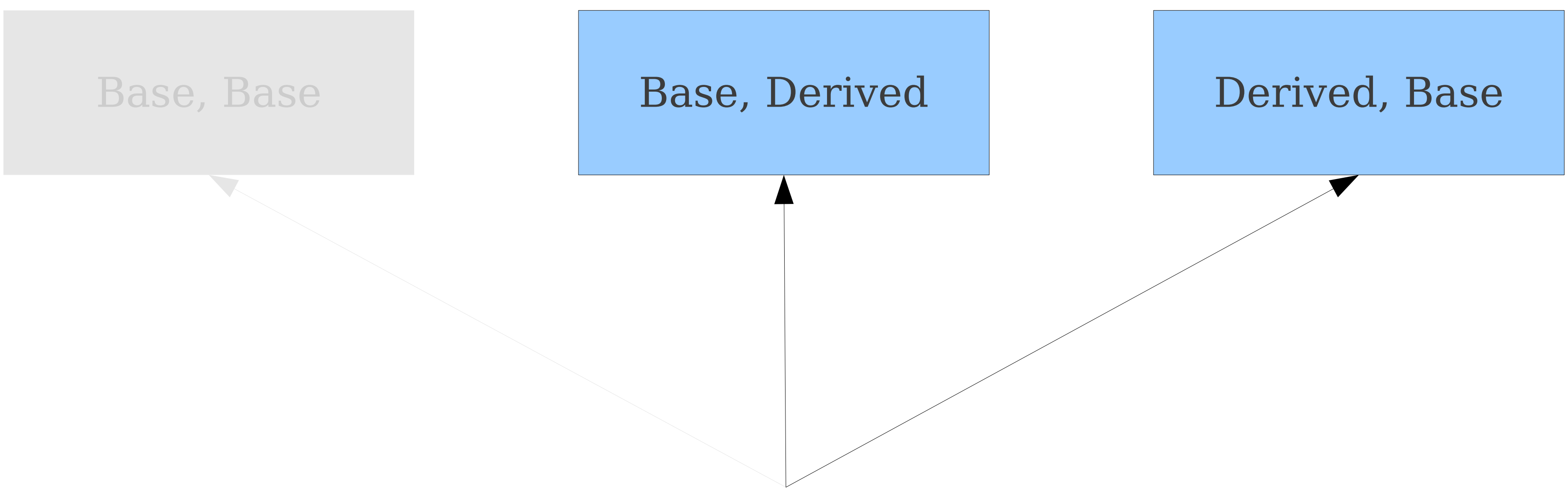
```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Base b1, Derived d2);
```



`Function(new Derived, new Derived);`

Ambiguous Calls

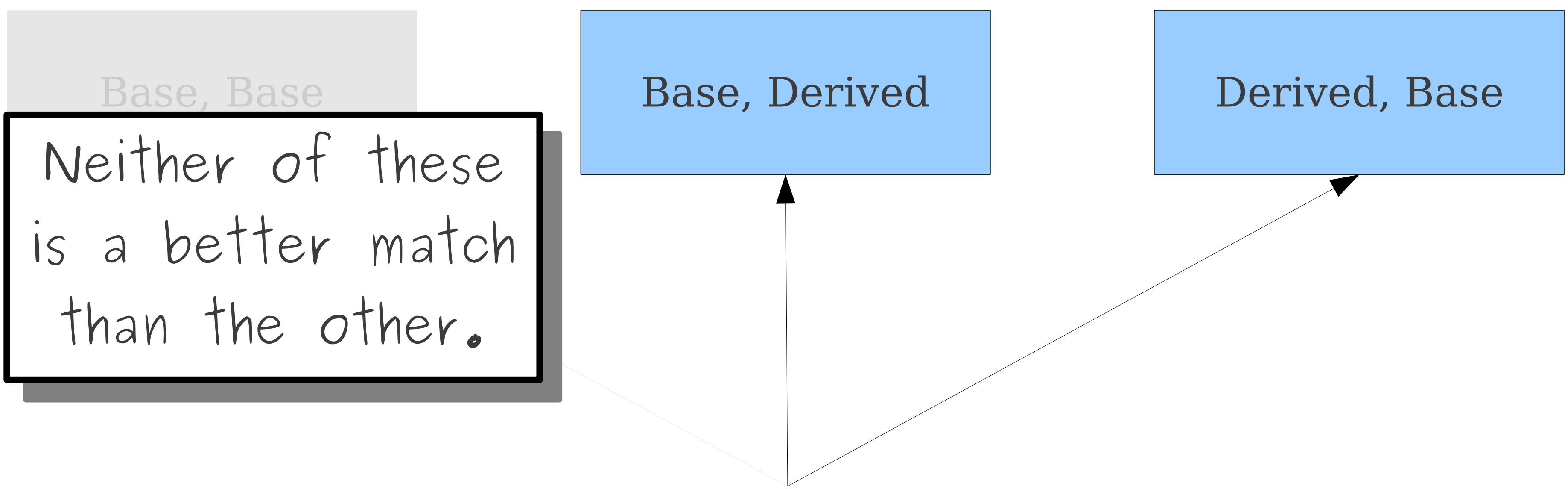
```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Base b1, Derived d2);
```




```
Function(new Derived, new Derived);
```

Ambiguous Calls

```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Base b1, Derived d2);
```




```
Function(new Derived, new Derived);
```

In the Real World

- Often much more complex than this.
- Example: **variadic functions**.
  - Functions that can take multiple arguments.
- Supported by C, C++, and Java.

Overloading with Variadic Functions

```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```

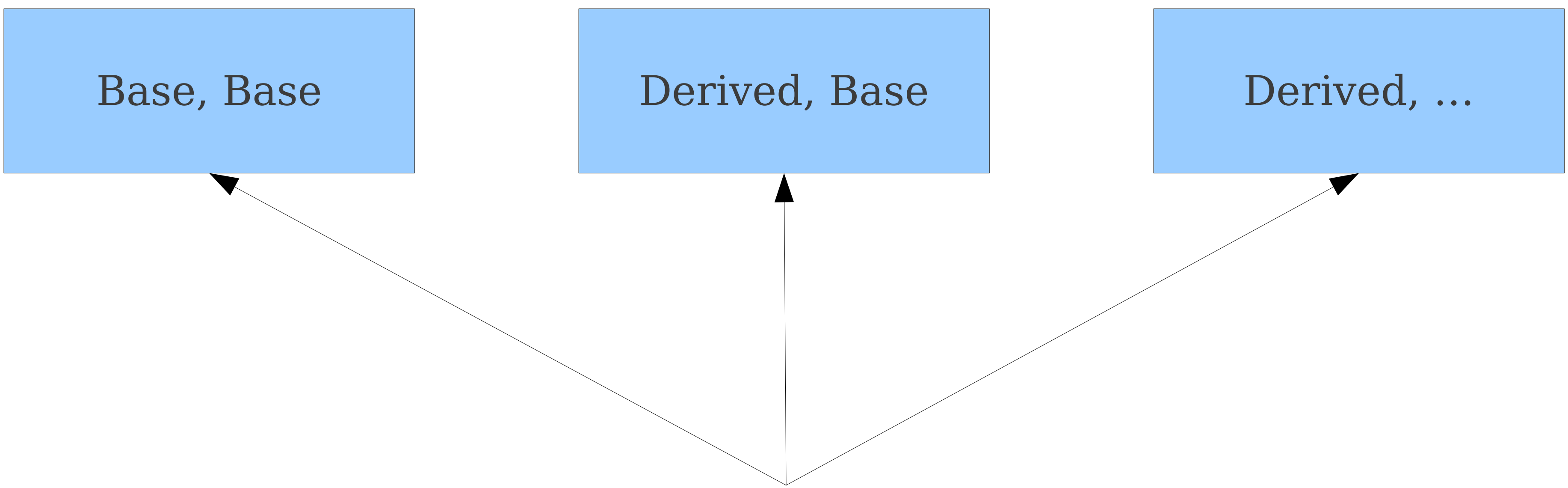
Overloading with Variadic Functions

```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```

```
Function(new Derived, new Derived);
```

Overloading with Variadic Functions

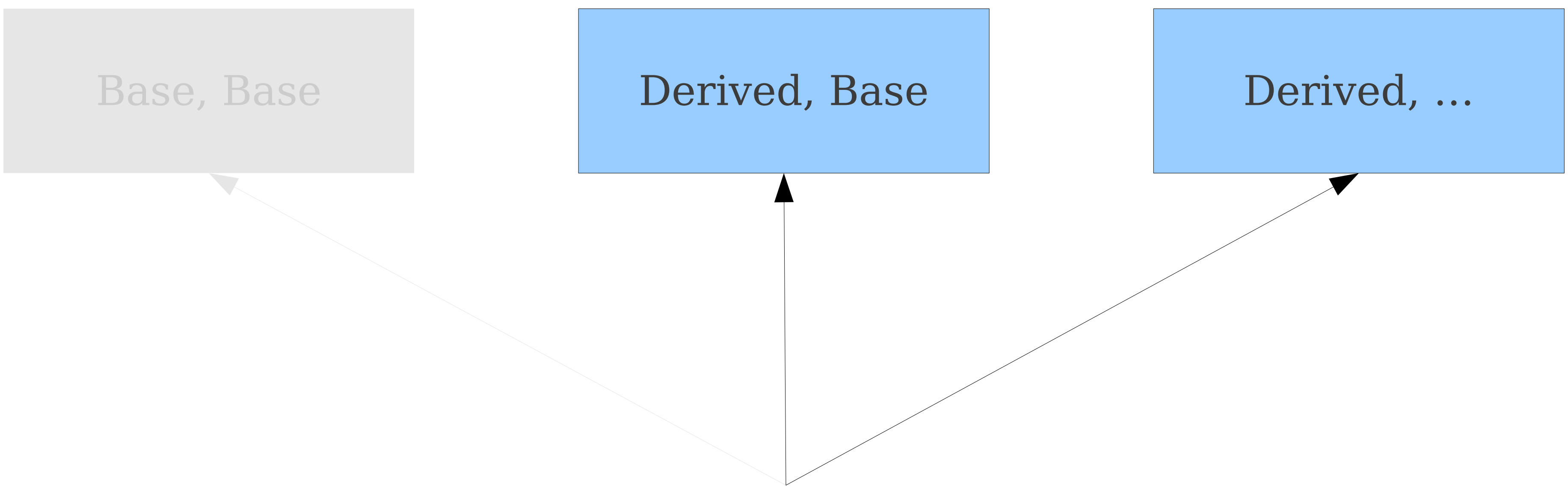
```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```



`Function(new Derived, new Derived);`

Overloading with Variadic Functions

```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```




```
Function(new Derived, new Derived);
```

Overloading with Variadic Functions

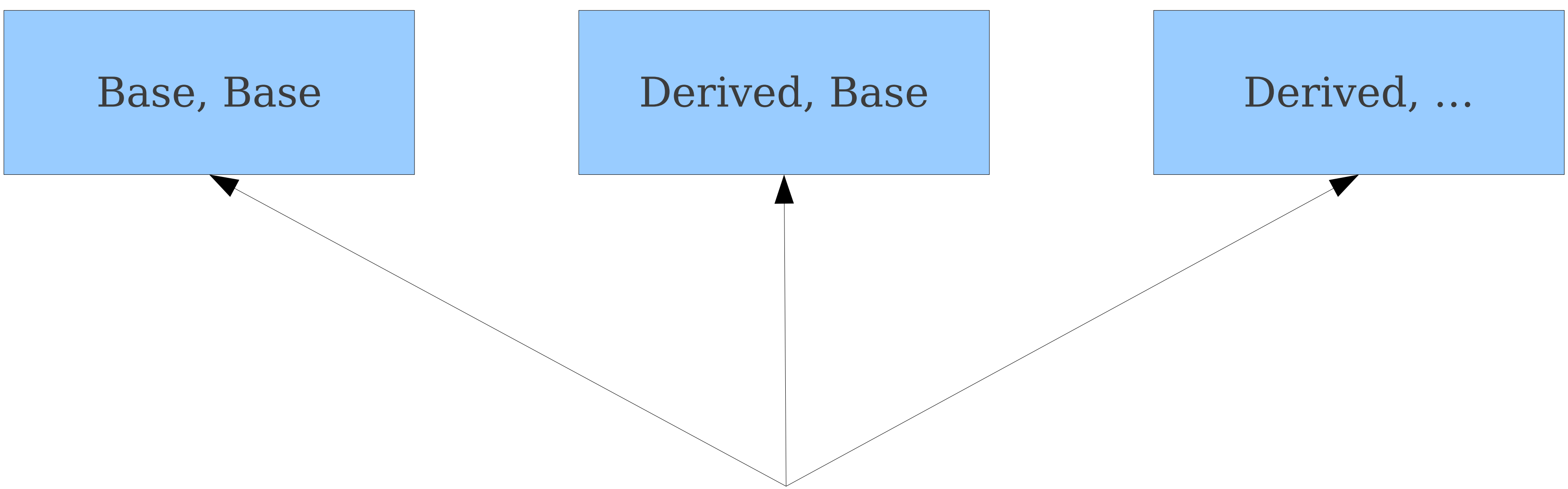
- Option one: **Consider the call ambiguous**.
  - There are indeed multiple valid function calls, and that's that!
- Option two: **Prefer the non-variadic function**.
  - A function specifically designed to handle a set of arguments is probably a better match than one designed to handle arbitrarily many parameters.
  - Used in both C++ and (with minor modifications) Java.

Hierarchical Function Overloads

- Idea: Have a hierarchy of candidate functions.
- Conceptually similar to a scope chain:
  - Start with the lowest hierarchy level and look for an overload.
  - If a match is found, choose it.
  - If multiple functions match, report an ambiguous call.
  - If no match is found, go to the next level in the chain.
- Similar techniques used in other places:
  - Template / generic functions.
  - Implicit conversions

Overloading with Variadic Functions

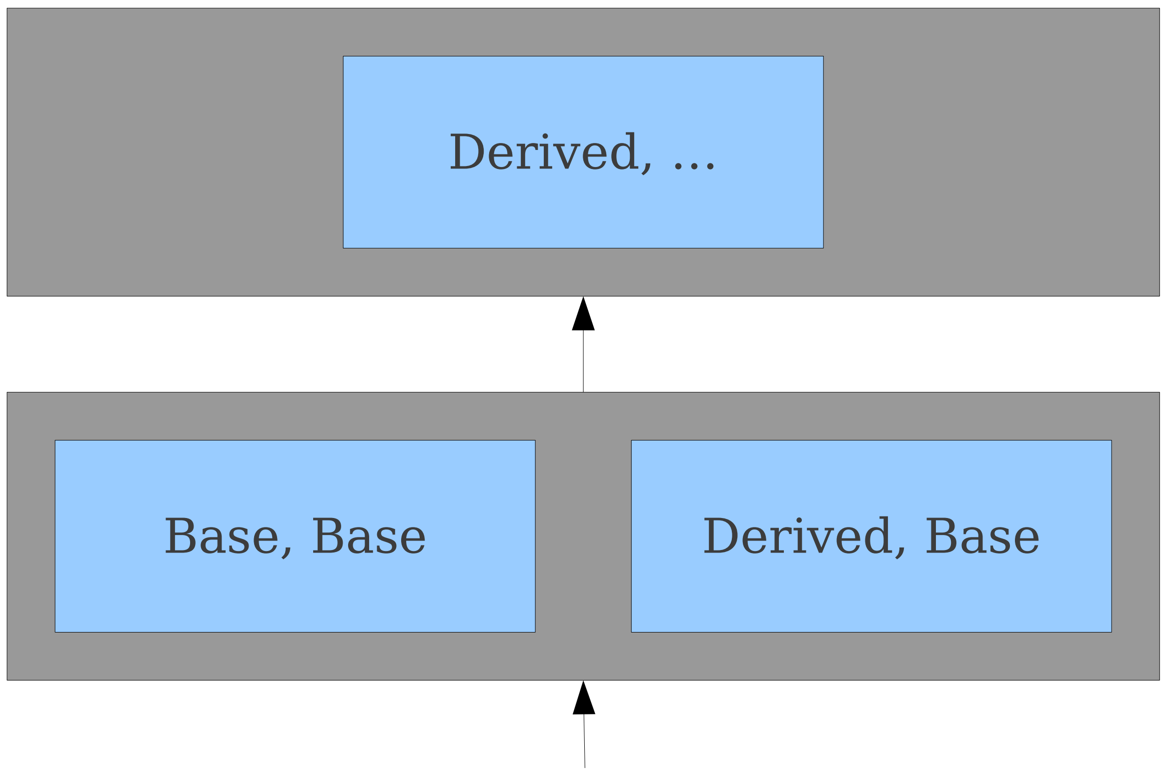
```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```




```
Function(new Derived, new Derived);
```

Overloading with Variadic Functions

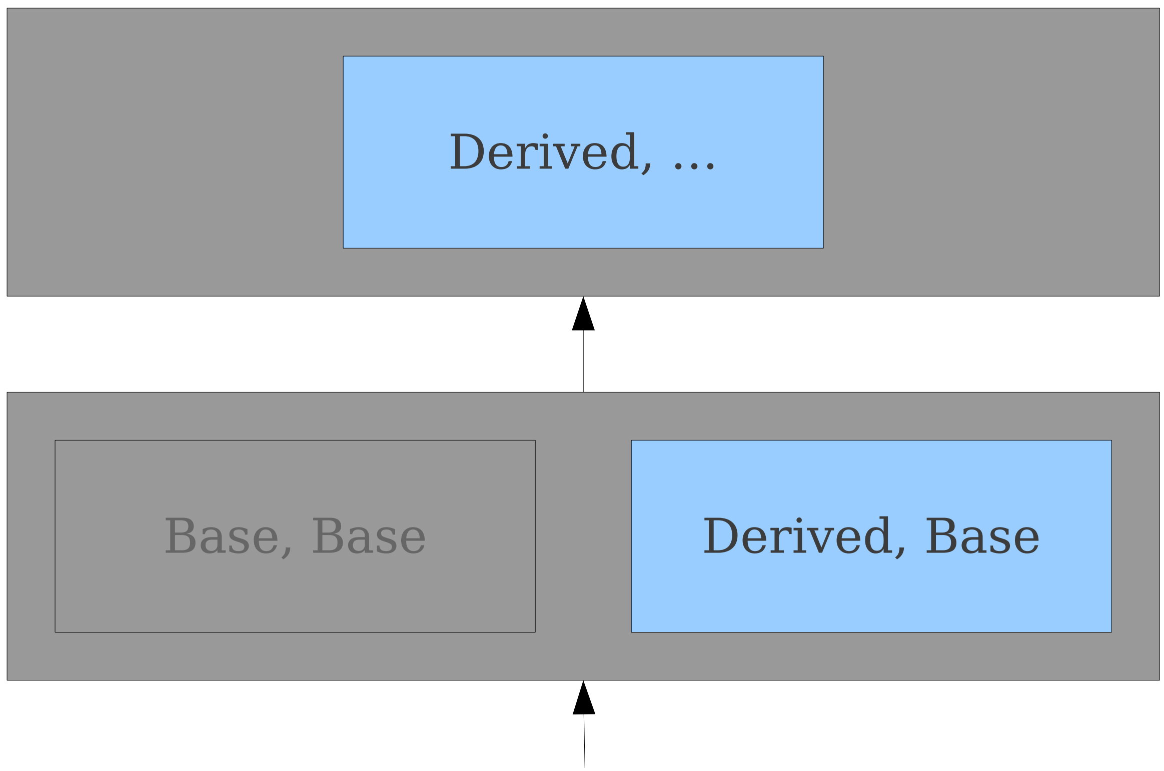
```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```



`Function(new Derived, new Derived);`

Overloading with Variadic Functions

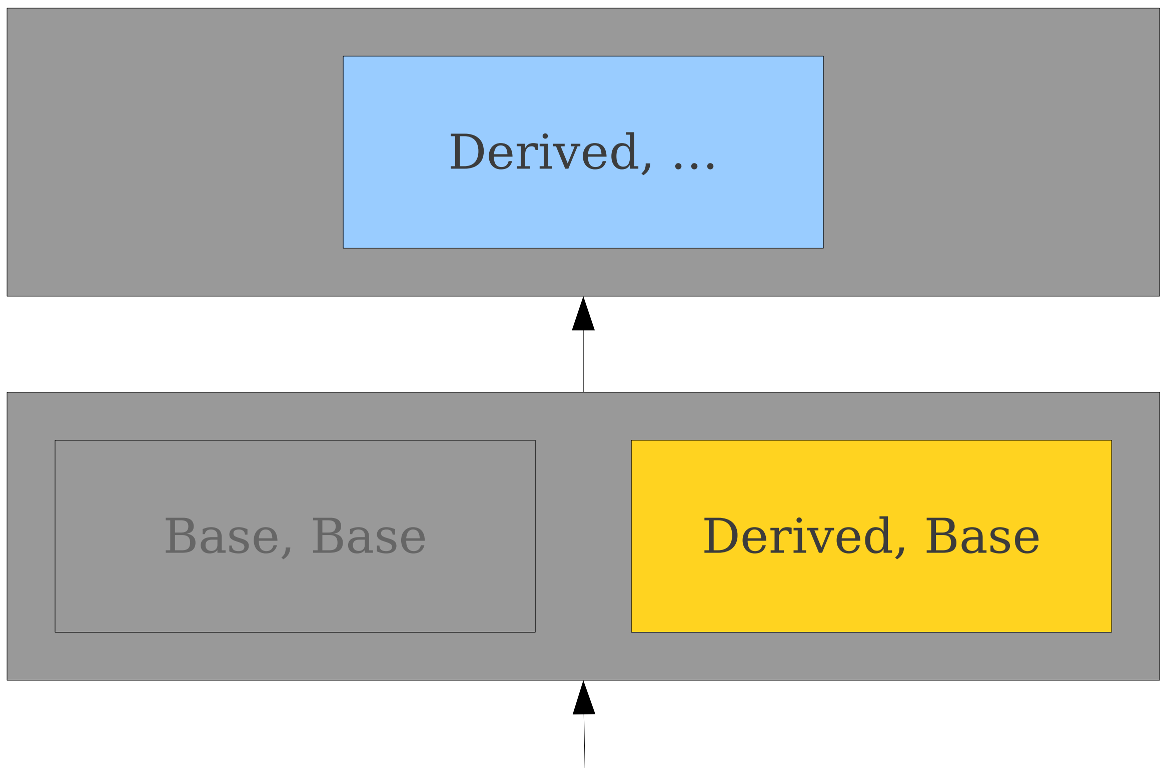
```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```




```
Function(new Derived, new Derived);
```

Overloading with Variadic Functions

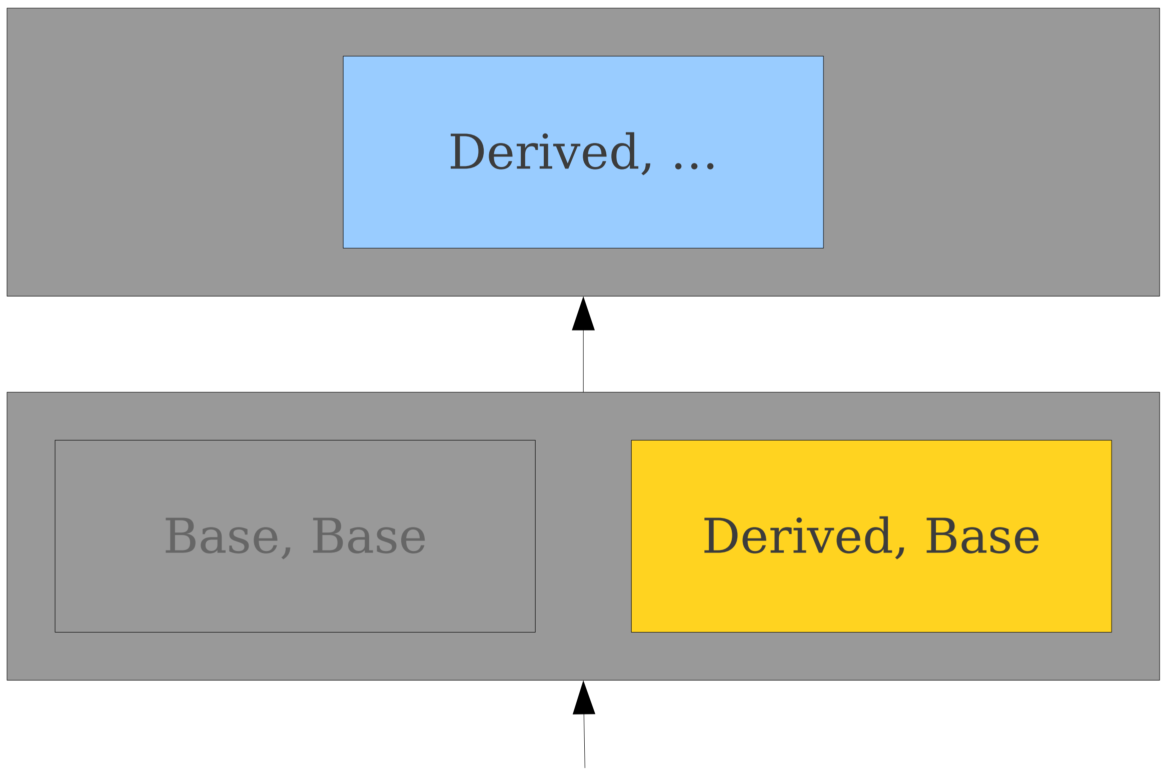
```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```




```
Function(new Derived, new Derived);
```

Overloading with Variadic Functions

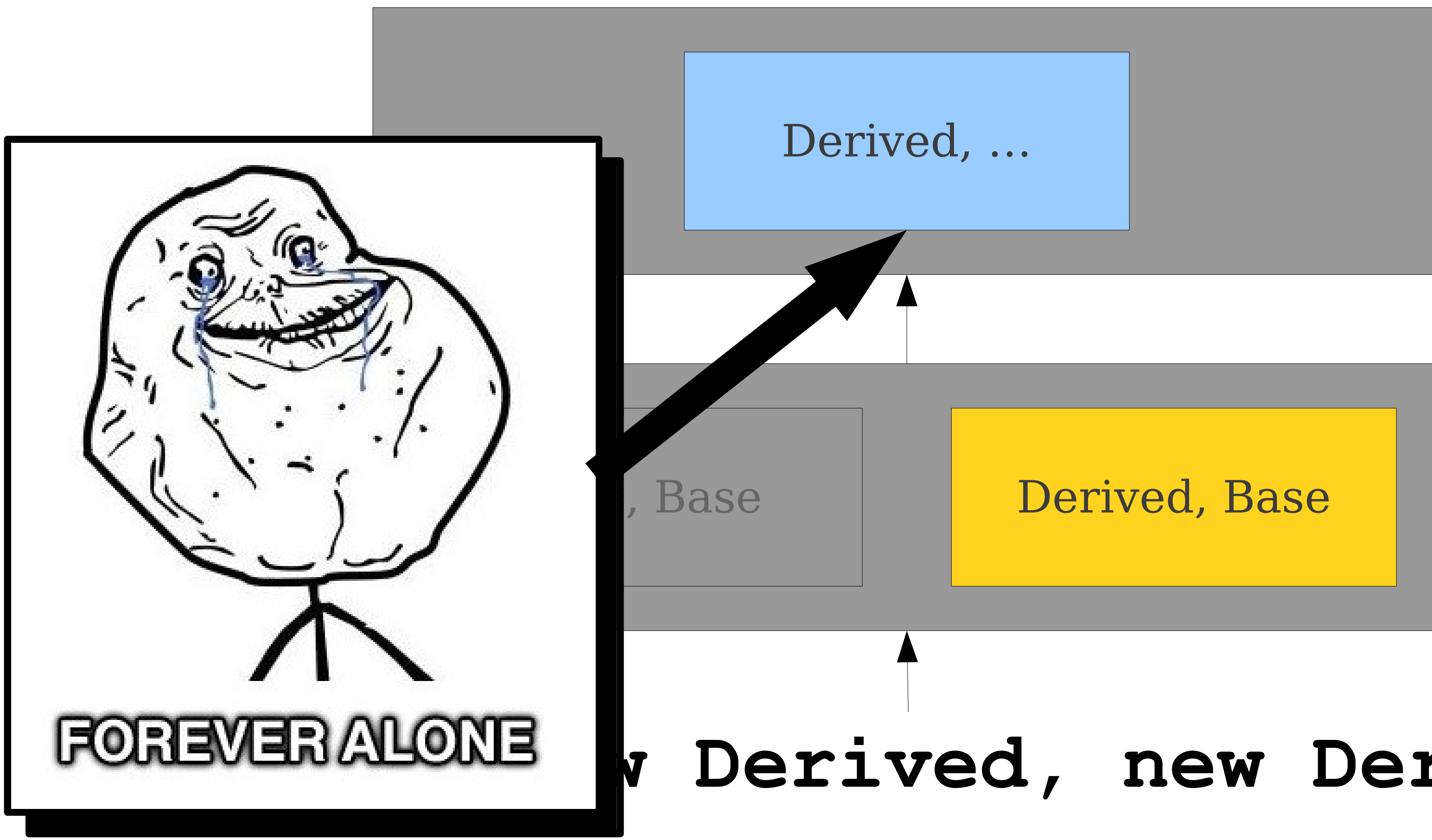
```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```



`Function(new Derived, new Derived);`

Overloading with Variadic Functions

```
void Function(Base b1, Base b2);
void Function(Derived d1, Base b2);
void Function(Derived d1, ...);
```




```
w Derived, new Derived);
```

Covariance and Contravariance

A Rule for Member Functions

$$\frac{}{S \vdash e_0.f(e_1, \ldots, e_n) \; : \; ?}$$

A Rule for Member Functions

f is an identifier.

$$\frac{}{S \vdash e_0.f(e_1, \ldots, e_n) \; : \; ?}$$

A Rule for Member Functions

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) \ : \ ?}$$

A Rule for Member Functions

f is an identifier.
$S \vdash e_0 : M$
f is a member function in class M.

$$\frac{}{S \vdash e_0.f(e_1, \ldots, e_n) \ : ?}$$

A Rule for Member Functions

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ f \text{ is a member function in class } M. \\ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) \ : ?}$$

A Rule for Member Functions

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ f \text{ is a member function in class M.} \\ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \\ S \vdash e_i : R_i \;\; \text{for } 1 \leq i \leq n \\ R_i \leq T_i \;\; \text{for } 1 \leq i \leq n \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) \;\; : ?}$$

A Rule for Member Functions

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ f \text{ is a member function in class } M. \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : R_i \text{ for } 1 \le i \le n \\ R_i \le T_i \text{ for } 1 \le i \le n \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) : U}$$

Legality and Safety

```
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* … */
    }
}

class Ego extends Id {
    void bePractical() {
        /* … */
    }
}

int main() {
    (new Ego).me().bePractical();
}
```

Legality and Safety

```
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* … */
    }
}

class Ego extends Id {
    void bePractical() {
        /* … */
    }
}

int main() {
    (new Ego).me().bePractical();
}
```

Legality and Safety

```
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* … */
    }
}

class Ego extends Id {
    void bePractical() {
        /* … */
    }
}

int main() {
    (new Ego).me().bePractical();
}
```

Ego

Legality and Safety

```
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* … */
    }
}

class Ego extends Id {
    void bePractical() {
        /* … */
    }
}

int main() {
    (new Ego).me().bePractical();
}
```

Ego

Legality and Safety

```
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* … */
    }
}

class Ego extends Id {
    void bePractical() {
        /* … */
    }
}

int main() {
    (new Ego).me().bePractical();
}
```

$\underbrace{\texttt{(new Ego)}}_{\textbf{Ego}}$

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ f \text{ is a member function in class } M. \\ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \\ S \vdash e_i : R_i \ \text{ for } 1 \leq i \leq n \\ R_i \leq T_i \ \text{ for } 1 \leq i \leq n \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) \ : U}$$

Legality and Safety

```
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* … */
    }
}

class Ego extends Id {
    void bePractical() {
        /* … */
    }
}

int main() {
    (new Ego).me().bePractical();
}
```

$\underbrace{\texttt{(new Ego)}}_{\textbf{Ego}}$

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ f \text{ is a member function in class } M. \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : R_i \ \text{ for } 1 \le i \le n \\ R_i \le T_i \ \text{ for } 1 \le i \le n \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) \ : U}$$

Legality and Safety

```
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* … */
    }
}

class Ego extends Id {
    void bePractical() {
        /* … */
    }
}

int main() {
    (new Ego).me().bePractical();
}
```

$$\underbrace{\underbrace{(\texttt{new Ego})}_{\texttt{Ego}}\texttt{.me()}}_{\texttt{Id}}\texttt{.bePractical();}$$

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ f \text{ is a member function in class } M. \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : R_i \;\; \text{for } 1 \le i \le n \\ R_i \le T_i \;\; \text{for } 1 \le i \le n \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) \;\; : U}$$

Legality and Safety

```
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* … */
    }
}

class Ego extends Id {
    void bePractical() {
        /* … */
    }
}

int main() {
    (new Ego).me().bePractical();
}
```

$$\underbrace{\underbrace{\texttt{(new Ego)}}_{\texttt{Ego}}\texttt{.me()}}_{\texttt{Id}}\texttt{.bePractical();}$$

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ f \text{ is a member function in class } M. \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : R_i \quad \text{for } 1 \le i \le n \\ R_i \le T_i \quad \text{for } 1 \le i \le n \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) \ : U}$$

Legality and Safety

```
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* … */
    }
}

class Ego extends Id {
    void bePractical() {
        /* … */
    }
}

int main() {
    (new Ego).me().bePractical();
}
```

$\underbrace{\underbrace{\texttt{(new Ego)}}_{\texttt{Ego}}\texttt{.me()}}_{\texttt{Id}}$

$$\dfrac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ \boxed{f \text{ is a member function in class } M.} \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : R_i \quad \text{for } 1 \le i \le n \\ R_i \le T_i \quad \text{for } 1 \le i \le n \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) \;:\; U}$$

bePractical is not in Id!

Limitations of Static Type Systems

- Static type systems are often **incomplete**.
  - There are valid programs that are rejected.
- Tension between the **static** and **dynamic** types of objects.
  - Static type is the type declared in the program source.
  - Dynamic type is the actual type of the object at runtime.

Soundness and Completeness

- Static type systems sometimes reject valid programs because they cannot prove the absence of a type error.
- A type system like this is called **incomplete**.
- Instead, try to prove for every expression that

$$\text{DynamicType(E)} \leq \text{StaticType(E)}$$

- A type system like this is called **sound**.

An Impossibility Result

- Unfortunately, for most programming languages, it is provably impossible to have a sound and complete static type checker.
- Intuition: Could build a program that makes a type error iff a certain Turing machine accepts a given string.
- Type-checking equivalent to solving the halting problem!

Building a Good Static Checker

- It is difficult to build a good static type checker.
  - Easy to have unsound rules.
  - Impossible to accept all valid programs.
- Goal: make the language as complete as possible with sound type-checking rules.

Relaxing our Restrictions

```
class Base {
    Base clone() {
        return new Base;
    }
}

class Derived extends Base {
    Base clone() {
        return new Derived;
    }
}
```

Relaxing our Restrictions

```
class Base {
    Base clone() {
        return new Base;
    }
}

class Derived extends Base {
    Base clone() {
        return new Derived;
    }
}
```

Relaxing our Restrictions

```
class Base {
    Base clone() {
        return new Base;
    }
}

class Derived extends Base {
    Derived clone() {
        return new Derived;
    }
}
```

Relaxing our Restrictions

```
class Base {
    Base clone() {
        return new Base;
    }
}

class Derived extends Base {
    Derived clone() {
        return new Derived;
    }
}
```

Is this safe?

The Intuition

```
Base b = new Base;
Derived d = new Derived;
```

The Intuition

```
Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
```

The Intuition

```
Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
```

The Intuition

```
Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
```

The Intuition

```
Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
```

The Intuition

```
Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
Derived d3 = d.clone();
```

The Intuition

```
Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
Derived d3 = d.clone();

Base reallyD = new Derived;
```

The Intuition

```
Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
Derived d3 = d.clone();

Base reallyD = new Derived;
Base b4 = reallyD.clone();
```

The Intuition

```
Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
Derived d3 = d.clone();

Base reallyD = new Derived;
Base b4 = reallyD.clone();
Derived d4 = reallyD.clone();
```

The Intuition

```
Base b = new Base;
Derived d = new Derived;

Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
Derived d3 = d.clone();

Base reallyD = new Derived;
Base b4 = reallyD.clone();
Derived d4 = reallyD.clone();
```

Is this Safe?

f is an identifier.

$S \vdash e_0 : M$

f is a member function in class M.

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

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

$R_i \leq T_i$ for $1 \leq i \leq n$

$$\frac{}{S \vdash e_0.f(e_1, \ldots, e_n) : U}$$

Is this Safe?

This refers to the **static type** of the function.

f is an identifier.
$S \vdash e_0 : M$
f is a member function in class M.
f has type $(T_1, \ldots, T_n) \rightarrow U$
$S \vdash e_i : R_i$ for $1 \le i \le n$
$R_i \le T_i$ for $1 \le i \le n$

$$\frac{}{S \vdash e_0.f(e_1, \ldots, e_n) : U}$$

Is this Safe?

f is an identifier.
$S \vdash e_0 : M$
f is a member function in class M.
f has type $(T_1, \ldots, T_n) \to U$
$S \vdash e_i : R_i$ for $1 \le i \le n$
$R_i \le T_i$ for $1 \le i \le n$

$$\frac{}{S \vdash e_0.f(e_1, \ldots, e_n) : U}$$

This refers to the **static** type of the function.

f has **dynamic** type

$(T_1, T_2, \ldots, T_n) \to V$

and we know that

$V \le U$

Is this Safe?

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ \boxed{\begin{array}{c} f \text{ is a member function in class M.} \\ f \text{ has type } (T_1, \ldots, T_n) \to U \end{array}} \\ S \vdash e_i : R_i \quad \text{for } 1 \le i \le n \\ R_i \le T_i \quad \text{for } 1 \le i \le n \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) \ : \boxed{U}}$$

This refers to the **static** type of the function.

f has **dynamic** type

$(T_1, T_2, \ldots, T_n) \to V$

and we know that

$\boxed{V \le U}$

So the rule is sound!

Covariant Return Types

- Two functions A and B are **covariant** in their return types if the return type of A is convertible to the return type of B.
- Many programming language support covariant return types.
  - C++ and Java, for example.
- Not supported in Decaf.
  - But easy extra credit!

Relaxing our Restrictions (Again)

```
class Base {
    bool equalTo(Base B) {
        /* … */
    }
}

class Derived extends Base {
    bool equalTo(Base B) {
        /* … */
    }
}
```

Relaxing our Restrictions (Again)

```
class Base {
    bool equalTo(Base B) {
        /* … */
    }
}

class Derived extends Base {
    bool equalTo(Base B) {
        /* … */
    }
}
```

Relaxing our Restrictions (Again)

```
class Base {
    bool equalTo(Base B) {
        /* … */
    }
}

class Derived extends Base {
    bool equalTo(Derived B) {
        /* … */
    }
}
```

Relaxing our Restrictions (Again)

```
class Base {
    bool equalTo(Base B) {
        /* … */
    }
}

class Derived extends Base {
    bool equalTo(Derived D) {
        /* … */
    }
}
```

Relaxing our Restrictions (Again)

```
class Base {
    bool equalTo(Base B) {
        /* … */
    }
}

class Derived extends Base {
    bool equalTo(Derived D) {
        /* … */
    }
}
```

Is this safe?

Is this Safe?

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ f \text{ is a member function in class M.} \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : R_i \ \text{ for } 1 \le i \le n \\ R_i \le T_i \ \text{ for } 1 \le i \le n \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) \ : U}$$

Is this Safe?

f is an identifier.
$S \vdash e_0 : M$
f is a member function in class M.
f has type $(T_1, \ldots, T_n) \to U$
$S \vdash e_i : R_i$ for $1 \leq i \leq n$
$R_i \leq T_i$ for $1 \leq i \leq n$

$$\frac{}{S \vdash e_0.f(e_1, \ldots, e_n) : U}$$

This refers to the **static type** of the function.

Is this Safe?

f is an identifier.
$S \vdash e_0 : M$
f is a member function in class M.
f has type $(T_1, \ldots, T_n) \to U$
$S \vdash e_i : R_i$ for $1 \leq i \leq n$
$R_i \leq T_i$ for $1 \leq i \leq n$

$$\frac{}{S \vdash e_0.f(e_1, \ldots, e_n) \;:\; U}$$

This refers to the **static** type of the function.

f has **dynamic** type

$(V_1, V_2, \ldots, V_n) \to U$

and we know that

$V_i \leq T_i$ for $1 \leq i \leq n$

Is this Safe?

f is an identifier.
$S \vdash e_0 : M$
f is a member function in class M.
f has type $(T_1, \ldots, T_n) \to U$
$S \vdash e_i : R_i$ for $1 \le i \le n$
$R_i \le T_i$ for $1 \le i \le n$

$$\frac{}{S \vdash e_0.f(e_1, \ldots, e_n) \ : U}$$

This refers to the **static** type of the function.

f has **dynamic** type

$(V_1, V_2, \ldots, V_n) \to U$

and we know that

$V_i \le T_i$ for $1 \le i \le n$

$R_i \le T_i$ for $1 \le i \le n$
$V_i \le T_i$ for $1 \le i \le n$

Is this Safe?

This refers to the **static** type of the function.

f is an identifier.
$S \vdash e_0 : M$
f is a member function in class M.
f has type $(T_1, \ldots, T_n) \to U$
$S \vdash e_i : R_i$ for $1 \le i \le n$
$R_i \le T_i$ for $1 \le i \le n$

$$\frac{}{S \vdash e_0.f(e_1, \ldots, e_n) \ : U}$$

f has **dynamic** type

$(V_1, V_2, \ldots, V_n) \to U$

and we know that

$V_i \le T_i$ for $1 \le i \le n$

$R_i \le T_i$ for $1 \le i \le n$
$V_i \le T_i$ for $1 \le i \le n$

This doesn't mean that
$R_i \le V_i$ for $1 \le i \le n$

A Concrete Example

A Concrete Example

```
class Fine {
    void nothingFancy(Fine f) {
        /* … do nothing … */
    }
}
```

A Concrete Example

```
class Fine {
    void nothingFancy(Fine f) {
        /* … do nothing … */
    }
}

class Borken extends Fine {
    int missingFn() {
        return 137;
    }
    void nothingFancy(Borken b) {
        Print(b.missingFn());
    }
}
```

A Concrete Example

```
class Fine {
    void nothingFancy(Fine f) {
        /* … do nothing … */
    }
}

class Borken extends Fine {
    int missingFn() {
        return 137;
    }
    void nothingFancy(Borken b) {
        Print(b.missingFn());
    }
}

int main() {
    Fine f = new Borken;
    f.nothingFancy(new Fine);
}
```

A Concrete Example

```
class Fine {
    void nothingFancy(Fine f) {
        /* … do nothing … */
    }
}

class Borken extends Fine {
    int missingFn() {
        return 137;
    }
    void nothingFancy(Borken b) {
        Print(b.missingFn());
    }
}

int main() {
    Fine f = new Borken;
    f.nothingFancy(new Fine);
}
```

A Concrete Example

```
class Fine {
    void nothingFancy(Fine f) {
        /* … do nothing … */
    }
}

class Borken extends Fine {
    int missingFn() {
        return 137;
    }
    void nothingFancy(Borken b) {
        Print(b.missingFn());
    }
}

int main() {
    Fine f = new Borken;
    f.nothingFancy(new Fine);
}
```

(That calls this one)

A Concrete Example

```
class Fine {
    void nothingFancy(Fine f) {
        /* … do nothing … */
    }
}

class Borken extends Fine {
    int missingFn() {
        return 137;
    }
    void nothingFancy(Borken b) {
        Print(b.missingFn());
    }
}

int main() {
    Fine f = new Borken;
    f.nothingFancy(new Fine);
}
```

(That calls this one)

Covariant Arguments are Unsafe

- Allowing subclasses to restrict their parameter types is **fundamentally unsafe**.
- Calls through base class can send objects of the wrong type down to base classes.
- This is why Java's **`Object.equals`** takes another **`Object`**.
- Some languages got this wrong.
  - Eiffel allows functions to be covariant in their arguments; can cause runtime errors.

Contravariant Arguments

```
class Super {}
class Base extends Super {
    bool equalTo(Base B) {
        /* … */
    }
}

class Derived extends Base {
    bool equalTo(Base B) {
        /* … */
    }
}
```

Contravariant Arguments

```
class Super {}
class Base extends Super {
    bool equalTo(Base B) {
        /* … */
    }
}

class Derived extends Base {
    bool equalTo(Base B) {
        /* … */
    }
}
```

Contravariant Arguments

```
class Super {}
class Base extends Super {
    bool equalTo(Base B) {
        /* … */
    }
}

class Derived extends Base {
    bool equalTo(Super B) {
        /* … */
    }
}
```

Contravariant Arguments

```
class Super {}
class Base extends Super {
    bool equalTo(Base B) {
        /* … */
    }
}

class Derived extends Base {
    bool equalTo(Super B) {
        /* … */
    }
}
```

Contravariant Arguments

```
class Super {}
class Base extends Super {
    bool equalTo(Base B) {
        /* … */
    }
}

class Derived extends Base {
    bool equalTo(Super B) {
        /* … */
    }
}
```

Is this safe?

Is this Safe?

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ f \text{ is a member function in class } M. \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : R_i \ \text{ for } 1 \leq i \leq n \\ R_i \leq T_i \ \text{ for } 1 \leq i \leq n \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) \ : U}$$

Is this Safe?

This refers to the **static type** of the function.

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ \boxed{\begin{array}{c} f \text{ is a member function in class } M. \\ f \text{ has type } (T_1, \ldots, T_n) \to U \end{array}} \\ S \vdash e_i : R_i \quad \text{for } 1 \le i \le n \\ R_i \le T_i \quad \text{for } 1 \le i \le n \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) \ : U}$$

Is this Safe?

f is an identifier.
$S \vdash e_0 : M$
f is a member function in class M.
f has type $(T_1, \ldots, T_n) \to U$
$S \vdash e_i : R_i$ for $1 \leq i \leq n$
$R_i \leq T_i$ for $1 \leq i \leq n$

$$\frac{f \text{ is an identifier.} \quad S \vdash e_0 : M \quad f \text{ is a member function in class M.} \quad f \text{ has type } (T_1, \ldots, T_n) \to U \quad S \vdash e_i : R_i \text{ for } 1 \leq i \leq n \quad R_i \leq T_i \text{ for } 1 \leq i \leq n}{S \vdash e_0.f(e_1, \ldots, e_n) : U}$$

This refers to the **static type** of the function.

f has **dynamic type**

$(V_1, V_2, \ldots, V_n) \to U$

and we know that

$T_i \leq V_i$ for $1 \leq i \leq n$

Is this Safe?

f is an identifier.
$S \vdash e_0 : M$
f is a member function in class M.
f has type $(T_1, \ldots, T_n) \to U$
$S \vdash e_i : R_i$ for $1 \le i \le n$
$R_i \le T_i$ for $1 \le i \le n$

$$\frac{}{S \vdash e_0.f(e_1, \ldots, e_n) : U}$$

This refers to the **static** type of the function.

f has **dynamic** type

$(V_1, V_2, \ldots, V_n) \to U$

and we know that

$T_i \le V_i$ for $1 \le i \le n$

$R_i \le T_i$ for $1 \le i \le n$
$T_i \le V_i$ for $1 \le i \le n$

Is this Safe?

f is an identifier.
$S \vdash e_0 : M$
f is a member function in class M.
f has type $(T_1, \ldots, T_n) \rightarrow U$
$S \vdash e_i : R_i$ for $1 \leq i \leq n$
$R_i \leq T_i$ for $1 \leq i \leq n$

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ f \text{ is a member function in class M.} \\ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \\ S \vdash e_i : R_i \text{ for } 1 \leq i \leq n \\ R_i \leq T_i \text{ for } 1 \leq i \leq n \end{array}}{S \vdash e_0.f(e_1, \ldots, e_n) : U}$$

This refers to the **static** type of the function.

f has **dynamic** type

$(V_1, V_2, \ldots, V_n) \rightarrow U$

and we know that

$T_i \leq V_i$ for $1 \leq i \leq n$

$R_i \leq T_i$ for $1 \leq i \leq n$
$T_i \leq V_i$ for $1 \leq i \leq n$
so
$\mathbf{R_i \leq V_i}$ **for** $\mathbf{1 \leq i \leq n}$

Contravariant Arguments are Safe

- Intuition: When called through base class, will accept anything the base class already would.
- Most languages do not support contravariant arguments.
- Why?
  - Increases the complexity of the compiler and the language specification.
  - Increases the complexity of checking method overrides.

Contravariant Overrides

```
class Super {}
class Duper extends Super {}
class Base extends Super {
    bool equalTo(Base B) {
        /* … */
    }
}

class Derived extends Base {
    bool equalTo(Super B) {
        /* … */
    }
    bool equalTo(Duper B) {
        /* … */
    }
}
```

Contravariant Overrides

```
class Super {}
class Duper extends Super {}
class Base extends Super {
    bool equalTo(Base B) {
        /* … */
    }
}

class Derived extends Base {
    bool equalTo(Super B) {
        /* … */
    }
    bool equalTo(Duper B) {
        /* … */
    }
}
```

Two overrides?
Or an over**load**
and an over**ride**?

So What?

- Need to be **very careful** when introducing language features into a statically-typed language.
- Easy to design language features; hard to design language features that are type-safe.
- Type proof system can sometimes help detect these errors in the abstract.

Summary

- We can extend our type proofs to handle well-formedness proofs.
- The **error type** is convertible to all other types and helps prevent cascading errors.
- Overloading is resolved at compile-time and determines which of many functions to call.
- Overloading ranks functions against one another to determine the best match.
- Functions can safely be **covariant** in their return types and **contravariant** in their argument types.

Next Time

- **Midterm Review Session**
  - Come with questions!
  - Leave with answers!