Principles of Programming Languages

Module M07: Type Systems

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

February 28 March 02, 07, 09 & 14, 2022

Table of Contents

Type Systems

Sources:

- *Concepts in Programming Languages* by John C. Mitchell, Cambridge University Press, 2003

What is a Type System?

- A **type system** is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute

Type Systems: Type & Type Error

Type Systems: Type & Type Error

What is a Type?

- In general, a **type** is a collection of computational entities that share some common property
  - *Type Examples*: *int*, *bool*, *int* $\rightarrow$ *bool*, etc.
  - *Type Non-Examples*: 3, *true*, *Even Integers*, etc.
  - Distinction between sets of values that are types and sets that are not types is language dependent
- There are three main uses of types in programming languages:
  - Naming and organizing concepts
  - Making sure that bit sequences in computer memory are interpreted consistently

```
1000001 => 65 if int
1000001 => 'A' if char
```

  - Providing information to the compiler about data manipulated by the program

Advantages of Types

- Program organization and documentation
  - Separate types for separate concepts
    - Represent concepts from problem domain
  - Document intended use of declared identifiers
    - Types can be checked, unlike program comments
- Identify and prevent errors
  - Compile-time or run-time checking can prevent meaningless computations such as `3 + true - "Bill"`
- Support optimization
  - Short integers require fewer bits
  - Access components of structures by known offset

What is a Type Error?

- A **type error** occurs when a computational entity, such as a function or a data value, is used in a manner that is inconsistent with the concept it represents
- Whatever the compiler/interpreter says it is?
- Something to do with bad bit sequences?
  - Floating point representation has specific form
  - An integer may not be a valid float
  - *Hardware Error*
    - ▷ `int x; x();`
    - ▷ `float_add(3, 4.5)`
- Something about programmer intent and use?
  - A type error occurs when a value is used in a way that is inconsistent with its definition
  - *Unintended Semantics*
    - ▷ `int_add(3, 4.5)`
    - ▷ declare as character, use as integer

Type errors are language dependent

- Array out of bounds access
  - C/C++: runtime errors
  - Haskell/Java: dynamic type errors
- Null pointer dereference
  - C/C++: run-time errors
  - Haskell/ML: pointers are hidden inside datatypes
    - Null pointer dereferences would be incorrect use of these datatypes, therefore static type errors

Type Systems: Type Safety

Type Systems: Type Safety

Type Safety

- A programming language is **type safe** if no program is allowed to violate its type distinctions
- A safe language protects its own abstractions:

Safety	*Statically checked*	*Dynamically checked*	*Remarks*
Unsafe	BCPL (*Basic Combined Programming Language*[1]) family including C, C++		• Type casts • Unions • Pointer arithmetic
Almost Safe	Algol family, Pascal, Ada		• Dangling pointers • Explicit Deallocation • Hard to make languages with explicit deallocation of memory fully type-safe
Safe	ML, Haskell, Java	Lisp, SmallTalk, Javascript, Scheme, Perl, Postscript, Python	• Complete type checking

[1] *Procedural, Imperative, and Structured programming language*

What are Type System good for?

- **Detecting Errors**: Static type-checking allows early detection of some programming errors
- **Abstraction**: Enforces disciplined programming
- **Documentation**: Types are useful when reading programs
- **Language Safety**: A safe language protects its own abstractions; Portability
- **Efficiency**: Distinguish between integer-valued arithmetic expressions and real-valued ones; Eliminate many of the dynamic checks; etc.
- **Other Applications**
  - *Computer and network security*
  - *Program analysis tools*
  - *Automated theorem proving*
  - *Database* – type analysis of *Document Type Definitions* and other kinds of schemas (such as XML-Schema standard [XS 2000]) for describing structured data in XML
  - *Computational linguistics* – typed λ–calculi form the basis for formalisms such as *categorical grammar*

Type Systems: Type Checking & Inference

Compile-time vs Run-time Checking

- JavaScript and Lisp use run-time type checking
  - `f(x)`: Make sure `f` is a function before calling `f`
  - ```
    $ var f = 3
    $ f(2);
    $ typein:3: TypeError: f is not a function
    ```
  - In JavaScript, we can write a function like
    ```
    function f(x) { return x < 10 ? x : x(); }
    ```
    Some uses will produce type error, some will not
- Haskell and Java use compile-time type checking
  - `f(x)`: Must have $f::A \to B$ and $x::A$ inside datatypes
- Basic tradeoff
  - Both kinds of checking prevent type errors
  - Run-time checking slows down execution
  - Compile-time checking restricts program flexibility
    - JavaScript array: elements can have different types
    - Haskell list: all elements must have same type
  - Which gives better programmer diagnostics?
    ```

Compile-time vs Run-time Checking: Conservativity of Compile-Time Checking

- A property of compile-time type checking is that the compiler must be conservative
- The Compile-time type checking will find all statements and expressions that produce run-time type errors, but also may flag statements or expressions as errors even if they do not produce run-time errors
- More specifically, most checkers are both *sound* and *conservative*
  - A type checker is *sound* if no programs with errors are considered correct
  - A type checker is *conservative* if some programs without errors are still considered to have errors
- For any Turing-complete programming language, the set of programs that may produce a run-time type error is undecidable

```
if (complicated-expression-that-could-run-forever)
    then (expression-with-type-error)
    else (expression-with-type-error)
```

- Static typing is always conservative

```
function f(x) { return x < 10 ? x : x(); }
if (complicated-boolean-expression)
    then f(5);
    else f(15);
```

Compile-time vs Run-time Checking: Comparative

- Trade-offs between compile-time and run-time checking

Form of Type Checking	Advantages	Disadvantages
Run-time	• Prevents type errors • Need not be conservative	• Slows program execution
Compile-time	• Prevents type errors • Eliminates run-time tests • Finds type errors before execution and run-time tests	• May restrict programming because tests are *conservative*

- Combining Compile-Time and Run-Time Checking
  - Most programming languages actually use some combination of compile-time and run-time type checking
  - In Java, for example, static type checking is used to distinguish arrays from integers, but array bounds errors (which are a form of type error) are checked at run-time

Type Inference

- **Type inference** is the process of determining the types of expressions based on the known types of some symbols that appear in them
- The difference between type inference and compile-time type checking is really a matter of degree
- A **type-checking** algorithm goes through the program to check that the types declared by the programmer agree with the language requirements
- In **type inference**, the idea is that some information is not specified, and some form of logical inference is required for determining the types of identifiers from the way they are used

Type Inference: Checking vs Inference

- Standard Type Checking

```
int f(int x) { return x+1; };
int g(int y) { return f(y+1)*2; };
```

  - Examine body of each function
  - Use declared types to check agreement

- Type Inference

```
T f(T x)  return x+1; ;       // T => char, int, long, float, double, ...
                              // any pointer type other than void*
                              // T => any type with operator+()
T g(T y)  return f(y+1)*2; ; // T => char, int, long, float, double, ...
                              // T => any type with operator*()
                              // T cannot be any pointer type
```

  - Examine code without type information
  - Infer the most general types that could have been declared

Type Inference

- In addition to providing a flexible form of compile-time type checking, type inference supports **polymorphism**
- Type-inference algorithm uses **type variables** as placeholders for types that are not known
- In some cases, the type-inference algorithm resolves all type variables and determines that they must be equal to specific types such as Int, Bool, or String
- In other cases, the type of a function may contain type variables that are not constrained by the way the function is defined. In these cases, the function may be applied to any arguments whose types match the form given by a type expression containing type variables

Type Inference

Sources:

- *Concepts in Programming Languages* by John C. Mitchell, Cambridge University Press, 2003
- *Lecture 26: Type Inference and Unification*, Cornell University, 2005

Type Inference by Hand-weaving: Example 1

- Consider

```
f x = 2 + x
> f :: Int -> Int
```

- What is the type of `f`?
  - `+` has type: `Int` $\rightarrow$ `Int` $\rightarrow$ `Int`
  - `2` has type: `Int`
  - Since we are applying `+` to `x` we need `x :: Int`
  - Therefore `f x = 2 + x` has type `Int` $\rightarrow$ `Int`

Type Inference by Hand-weaving: Example 2

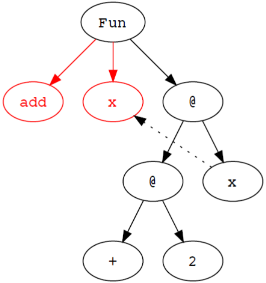
- Consider

```
f (g, h) = g (h(0))
> f :: (a -> b, Int -> a) -> b
```

- What is the type of `f`?
  - `h` is applied to an integer argument, and so `h` must be a function from `Int` to something
  - Represent "something" by introducing a type variable `a`
  - `g` must be a function that takes whatever `h` returns (of type `a`) and then returns something else
  - `g` is not constrained to return the same type of value as `h`, so we represent this second one by a new type variable, `b`
  - Putting the types of `h` and `g` together, we see that the first argument to f has type ($a \rightarrow b$) and the second has type ($Int \rightarrow a$)
  - Function `f` takes the pair of these two functions as an argument and returns the same type of value as `g` returns
  - Therefore, the type of `f` is ($a \rightarrow b$, $Int \rightarrow a$) $\rightarrow b$

Type Inference Algorithm: Example 1: Simple Function: `add x = 2 + x`

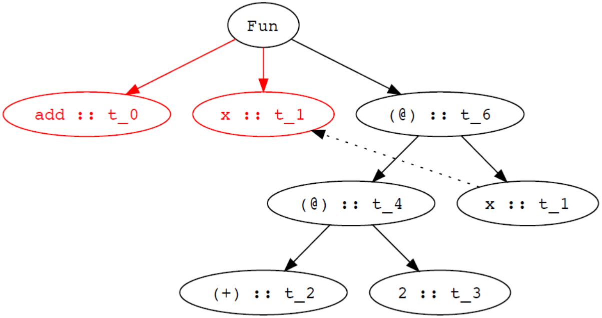
- `add x = 2 + x`
  `add :: ?`
- **Parse Program** text to construct parse tree
- Infix operators are converted to Curried function application during parsing: `2 + x` ⇒ `(+) 2 x`
- Dotted link shows where the variable is bound



Parse Tree for Add Function

Type Inference Algorithm: Example 1: Simple Function: `add x = 2 + x`

- `add x = 2 + x`
- **Assign type variables to nodes**
- Variables are given same type as binding occurrence

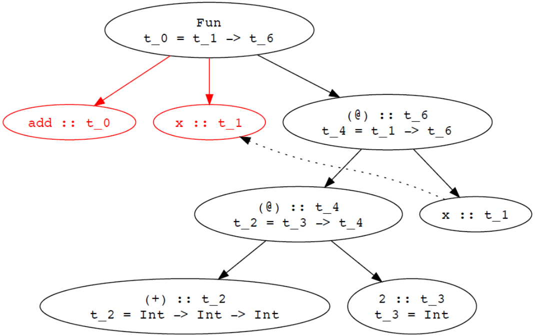


Parse Tree Labeled with Type Variables

Type Inference Algorithm: Example 1: Simple Function: `add x = 2 + x`

- **Add Constraints**

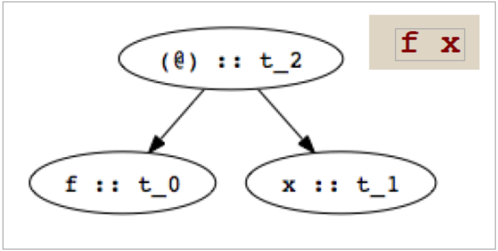
```
t_0 = t_1 -> t_6
t_4 = t_1 -> t_6
t_2 = t_3 -> t_4
t_2 = Int -> Int -> Int
t_3 = Int
```



Parse Tree Labeled with Type Constraints

Type Inference Algorithm: Constraints from Application Nodes

- Function application (apply f to x)
  - Type of f (t_0 in figure) must be *domain* → *range*
  - Domain of f must be type of argument x (t_1 in fig)
  - Range of f must be result of application (t_2 in fig)
  - Constraint: `t_0 = t_1 -> t_2`

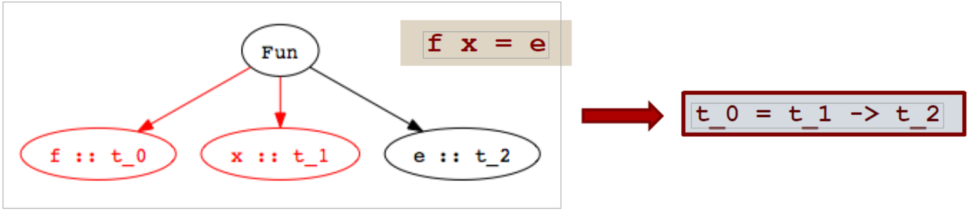


`t_0 = t_1 -> t_2`

- We shall see this formally in Slides 51

Type Inference Algorithm: Constraints from Abstractions

- Function declaration / abstraction
  - Type of f (t_0 in figure) must be *domain* → *range*
  - Domain is type of abstracted variable x (t_1 in fig)
  - Range is type of function body e (t_2 in fig)
  - Constraint: `t_0 = t_1 -> t_2`



- We shall see this formally in Slides 51

Type Inference Algorithm: Example 1: Simple Function: add x = 2 + x

- **Solve Constraints**

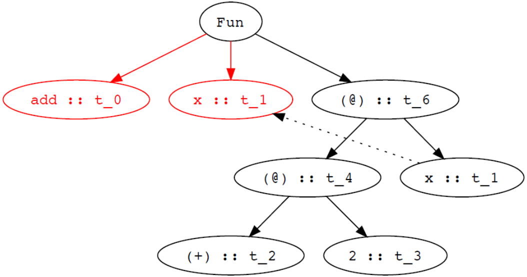
```
t_0 = t_1 -> t_6
t_4 = t_1 -> t_6
t_2 = t_3 -> t_4          ==>   t_3 -> t_4 = Int -> (Int -> Int)
t_2 = Int -> Int -> Int         t_3 = Int
t_3 = Int                       t_4 = Int -> Int

t_0 = t_1 -> t_6
t_4 = t_1 -> t_6          ==>   t_1 -> t_6 = Int -> Int
t_4 = Int -> Int                t_1 = Int
t_2 = Int -> Int -> Int         t_6 = Int
t_3 = Int

t_0 = Int -> Int
t_1 = Int
t_6 = Int
t_4 = Int -> Int
t_2 = Int -> Int -> Int
t_3 = Int
```

Type Inference Algorithm: Example 1: Simple Function: add x = 2 + x

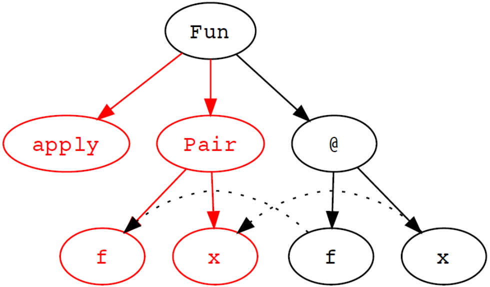
- **Determine type of declaration**
- ```
  t_0 = Int -> Int
  t_1 = Int
  t_6 = Int
  t_4 = Int -> Int
  t_2 = Int -> Int -> Int
  t_3 = Int
  ```
- ```
  add x = 2 + x
  > add :: Int -> Int
  ```



Parse Tree Labeled with Type Variables

Type Inference Algorithm: Example 2: Polymorphic Function: `apply (f, x) = f x`

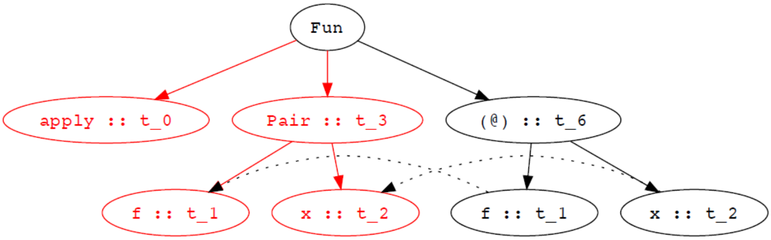
- `apply (f, x) = f x`
  `apply :: (t -> t1, t) -> t1`
- The `apply` function has a type involving type variables, making the function polymorphic
- **Parse Program** text to construct parse tree



Parse Tree for Apply Function

Type Inference Algorithm: Example 2: Polymorphic Function: `apply (f, x) = f x`

- `apply (f, x) = f x`
- **Assign type variables to nodes**

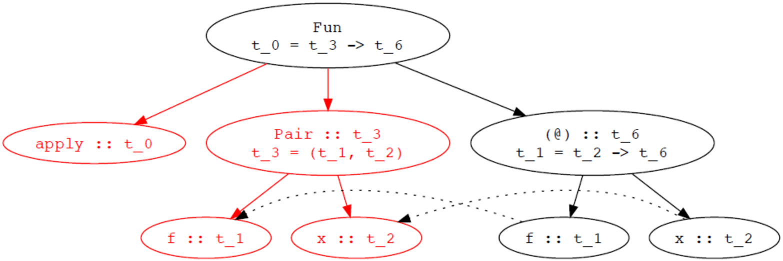


Parse Tree for Apply Function Labeled with Type Constraints

Type Inference Algorithm: Example 2: Polymorphic Function: `apply (f, x) = f x`

- **Add Constraints**

```
t_1 = t_2 -> t_6
t_0 = t_3 -> t_6
t_3 = (t_1, t_2)
```



Type Constraints for Apply Function

Type Inference Algorithm: Example 2: Polymorphic Function: apply (f, x) = f x

- **Solve Constraints**

```
t_0 = t_3 -> t_6
t_1 = t_2 -> t_6
t_3 = (t_1, t_2)

Replace t_3:
t_0 = (t_1, t_2) -> t_6
t_1 = t_2 -> t_6
t_3 = (t_1, t_2)

Replace t_1:
t_0 = (t_2 -> t_6, t_2) -> t_6
t_1 = t_2 -> t_6
t_3 = (t_2 -> t_6, t_2)

Replace t_2 w/ t and t_6 w/ t1:
t_0 = (t -> t1, t) -> t1
t_1 = t -> t1
t_3 = (t -> t1, t)
```

- **Determine Type**

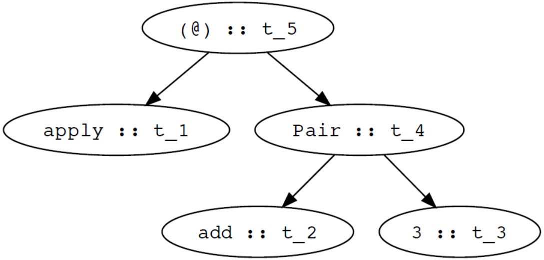
```
apply (f, x) = f x
> apply :: (t -> t1, t) -> t1
```

Type Inference Algorithm: Example 3: Function Application: apply (add, 3)


- ```
  apply (f, x) = f x
  > apply :: (a_1 -> a_2, a_1) -> a_2

  add x = 2 + x
  > add :: Int -> Int

  apply (add, 3) :: ?
  ```
- **Parse Tree and Assignment of Type Variables**

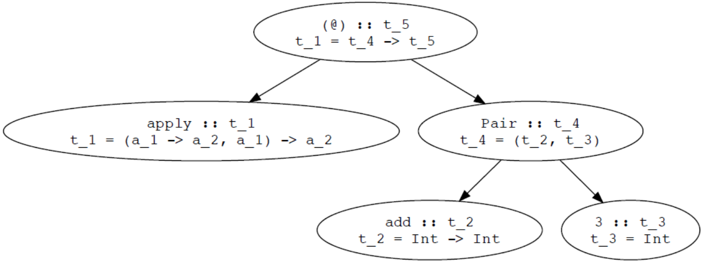


Type Variable Assignment for Apply Function Application Parse Tree
  ```

Type Inference Algorithm: Example 3: Function Application: apply (add, 3)

- **Add Constraints**

```
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_3 = Int
t_4 = (t_2, t_3)
```



Constraints for Apply Function Application Parse Tree

Type Inference Algorithm: Example 3: Function Application: apply (add, 3)

- **Solve Constraints**

```
Equations:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_3 = Int
t_4 = (t_2, t_3)
```

```
Equate t_1:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_3 = Int
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
```

```
Solution:
t_1 = (Int -> Int, Int) -> Int
t_2 = Int -> Int
t_3 = Int
t_4 = (Int -> Int, Int)
t_5 = Int
a_1 = Int
a_2 = Int
```

```
Equate t_4:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_2 = a_1 -> a_2
t_3 = Int
t_3 = a_1
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
```

```
Equate t_2:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_2 = a_1 -> a_2
t_3 = Int
t_3 = a_1
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
a_1 = Int
a_2 = Int
```

Type Inference Algorithm: Example 3: Function Application: apply (add, 3)

- **Determine Type**
- `apply (f, x) = f x`
  `> apply :: (a_1 -> a_2, a_1) -> a_2`

  ```
  add x = 2 + x
  > add :: Int -> Int

  apply (add, 3) :: t_5 = Int
  ```

Type Inference Algorithm: Example 4: Function Application: `apply (not, False)`


- ```
  apply (f, x) = f x
  > apply :: (a_1 -> a_2, a_1) -> a_2

  > not :: Bool -> Bool

  apply (not, False) :: ?
  ```
- Proceeding similarly as Example 4 gives:
  ```
  Solution:
  t_1 = (Bool -> Bool, Bool) -> Bool
  t_2 = Bool -> Bool
  t_3 = Bool
  t_4 = (Bool -> Bool, Bool)
  t_5 = Bool
  a_1 = Bool
  a_2 = Bool
  ```
- This fact illustrates the polymorphism of `apply`: Because the type `(a_1 -> a_2, a_1) -> a_2` of `apply` contains type variables, the function may be applied to any type of arguments that can be obtained if the type variables in `(a_1 -> a_2, a_1) -> a_2` are replaced with type names or type expressions
  ```

Type Inference Algorithm: Example 4A: Function Application: apply (add, False)


- ```
  apply (f, x) = f x
  > apply :: (a_1 -> a_2, a_1) -> a_2

  add x = 2 + x
  > add :: Int -> Int

  apply (add, False) :: ?
  ```
  ```

Type Inference Algorithm: Example 4A: Function Application: apply (add, False)

- **Solve Constraints**

```
Equations:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int


t_3 = Bool
t_4 = (t_2, t_3)
```

```
Equate t_4:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_2 = a_1 -> a_2
t_3 = Bool
t_3 = a_1
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
```

```
Equate t_1:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int


t_3 = Bool
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
```

```
Equate t_2:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_2 = a_1 -> a_2
t_3 = Bool
t_3 = a_1
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
a_1 = Int
a_2 = Int
```

```
Type Inconsistency:
t_1 = (Int -> Int, Int) -> Int
t_2 = Int -> Int
t_3 = Int => t_3 = a_1 = Int
             // Type Error
t_3 = Bool
t_4 = (Int -> Int, ?)
t_5 = Int
a_1 = Int
a_2 = Int
```

Type Inference Algorithm: Example 4B: Function Application: apply (not, 3)


- ```
  apply (f, x) = f x
  > apply :: (a_1 -> a_2, a_1) -> a_2

  > not :: Bool -> Bool

  apply (not, 3) :: ?
  ```
  ```

Type Inference Algorithm: Example 4B: Function Application: apply (not, 3)

- **Solve Constraints**

```
Equations:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Bool -> Bool
t_3 = Int

t_4 = (t_2, t_3)
```

```
Equate t_1:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Bool -> Bool
t_3 = Int

t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
```

```
Type Inconsistency:
t_1 = (Int -> Int, Int) -> Int
t_2 = Bool -> Bool
t_3 = Int
t_3 = Bool => t_3 = a_1 = Bool
             // Type Error
t_4 = (Bool -> Bool, ?)
t_5 = Bool
a_1 = Bool
a_2 = Bool
```

```
Equate t_4:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Bool -> Bool
t_2 = a_1 -> a_2
t_3 = Int
t_3 = a_1
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
```

```
Equate t_2:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Bool -> Bool
t_2 = a_1 -> a_2
t_3 = Int
t_3 = a_1
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
a_1 = Bool
a_2 = Bool
```

Type Inference: Inference Algorithm

Type Inference: Inference Algorithm

Type Inference Algorithm

[1] Assign a type to the expression and each sub-expression

- For any compound expression or variable, use a type variable
- For known operations or constants, such as $+$ or 3, use the type that is known for this symbol

[2] Generate a set of constraints on types, using the parse tree of the expression

- These constraints reflect the fact that if a function is applied to an argument, for example, then the type of the argument must equal the type of the domain of the function

[3] Solve these constraints by means of *unification*

- It is a substitution-based algorithm for solving systems of equations

Source: *Lecture 26: Type Inference and Unification*, Cornell University, 2005

Framing & Solving Type Constraints: Matrix Example

- Let us write the type of an $m \times n$ matrix as $m \to n$. Now the rules of matrix algebra can be expressed as typing rules:
  - Multiplication

$$\frac{\mathcal{E} \vdash A : s \to t, \mathcal{E} \vdash B : t \to u}{\mathcal{E} \vdash AB : s \to u}$$

  - Addition

$$\frac{\mathcal{E} \vdash A : s \to t, \mathcal{E} \vdash B : s \to t}{\mathcal{E} \vdash A + B : s \to t}$$

  - Squaring

$$\frac{\mathcal{E} \vdash A : s \to s}{\mathcal{E} \vdash A^2 : s \to s}$$

- What is the type of $(AB + CD)^2$, if the types of A, B, C, D are:

$$\begin{array}{lcl} A & : & s \to t \\ B & : & u \to v \\ C & : & w \to x \\ D & : & y \to z \end{array}$$

Framing & Solving Type Constraints: Matrix Example

- What is the type of $(AB + CD)^2$, if the types of A, B, C, D are:
  $A : s \rightarrow t, B : u \rightarrow v, C : w \rightarrow x, D : y \rightarrow z$
- We assign type variables for sub-expressions of $(AB + CD)^2$:

$$\begin{array}{lcl} \mathrm{A} & : & \mathrm{s} \rightarrow \mathrm{t} \\ \mathrm{B} & : & \mathrm{u} \rightarrow \mathrm{v} \\ \mathrm{C} & : & \mathrm{w} \rightarrow \mathrm{x} \\ \mathrm{D} & : & \mathrm{y} \rightarrow \mathrm{z} \end{array} \qquad \begin{array}{rcl} \mathrm{AB} & : & \mathrm{a} \rightarrow \mathrm{b} \\ \mathrm{CD} & : & \mathrm{c} \rightarrow \mathrm{d} \\ \mathrm{AB + CD} & : & \mathrm{e} \rightarrow \mathrm{f} \\ \mathrm{(AB + CD)^2} & : & \mathrm{g} \rightarrow \mathrm{h} \end{array}$$

- Applying the typing rules we get:

$$\begin{array}{lll} \mathrm{t = u,\ a = s,\ b = v} & \text{for} & AB \\ \mathrm{x = y,\ c = w,\ d = z} & \text{for} & CD \\ \mathrm{a = c = e,\ b = d = f} & \text{for} & AB + CD \\ \mathrm{e = f = g = h} & \text{for} & (AB + CD)^2 \end{array}$$

- Solving the constraints, we get three equivalence classes:
  $\mathrm{a = b = c = d = e = f = g = h = s = v = w = z}$
  $\mathrm{t = u}$
  $\mathrm{x = y}$
- Hence, $\begin{array}{lcl} \mathrm{A} & : & \mathrm{a} \rightarrow \mathrm{t} \\ \mathrm{B} & : & \mathrm{t} \rightarrow \mathrm{a} \\ \mathrm{C} & : & \mathrm{a} \rightarrow \mathrm{x} \\ \mathrm{D} & : & \mathrm{x} \rightarrow \mathrm{a} \end{array}$ is the most general typing. Any values for a, t, and x make the expression $(AB + CD)^2$ well-formed

Type Inference: Inference Algorithm: Unification & mgu

Unification

- The task of *unification* is to find a *substitution* S that **unifies** two given terms (that is, makes them equal). Let us write $s\ S$ for the result of applying the substitution S to the term s.
- Given s and t, we want to find S such that $s\ S = t\ S$. Such a substitution S is called a **unifier** for s and t.
- Example, given the two terms

```
f x (g y)            f (g z) w
```

where x, y, z, and w are variables, the substitution

```
S  =  [x <- g z, w <- g y]
```

would be a unifier, since

```
f x (g y) [x <- g z, w <- g y] =
f (g z) (g y) =
f (g z) w [x <- g z, w <- g y]
```

- *Unification is a purely syntactic definition; the meaning of expressions is not considered when computing unifiers*

Unification

- **Unifiers do not necessarily exist**. For example, the terms x and $f\ x$ cannot be unified, since no substitution for x can make the two terms equal.
- **Even when unifiers exist, they are not necessarily unique**. For example, for the two terms

```
f x (g y)          f (g z) w
```

the substitution

```
T  =  [x <- g (f a b),
       y <- f b a,
       z <- f a b,
       w <- g (f b a)]
```

is also a unifier:

```
f x (g y) T  =
f (g (f a b)) (g (f b a)) =
f (g z) w) T
```

Unification: mgu

- When a unifier exists, there is a **most general unifier (mgu)** that is unique up to renaming. A unifier S for s and t is an mgu for s and t if
  - S is a unifier for s and t; and
  - any other unifier T for s and t is a refinement of S; that is, T can be obtained from S by doing further substitutions.

  For example, the substitution

  ```
  S  =  [x <- g z, w <- g y]
  ```

  in the example above is an mgu for `f x (g y)` and `f (g z) w`. The unifier

  ```
  T  =  [x <- g (f a b), y <- f b a, z <- f a b, w <- g (f b a)]
  ```

  is a refinement of S, since $T = S\ U$, where

  ```
  U  =  [z <- f a b, y <- f b a]
  ```

  Note that

  ```
  f x (g y) S U
  = f x (g y) [x <- g z, w <- g y] [z <- f a b, y <- f b a]
  = f (g z) (g y) [z <- f a b, y <- f b a]
  = f (g (f a b)) (g (f b a))
  = f x (g y) T
  ```

Type Inference

- We assume that all bound variables are distinct. If not, we can rename bound variables (by α-reduction) to make this true. As in λ- calculus, it is always fine to α-convert. For example, `f x = x + 3` and `f y = y + 3` are semantically equivalent.
- The typing rules are:
  - **Function Application** (See Slide 26): An expression $(e_1\ e_2)$ only makes sense if e_1 is a function having a type of the form $s \rightarrow t$, and the input type of e_1 is the same as the type of its argument e_2. When these premises are satisfied, then the result, represented by the expression $(e_1\ e_2)$, has the same type as the result type of e_1.

$$\frac{\mathcal{E} \vdash e_1 : s \rightarrow t, \mathcal{E} \vdash e_2 : s}{\mathcal{E} \vdash (e_1\ e_2) : t}$$

  - **Function Abstraction** (See Slide 27): An expression $f\ x = e$ represents a function taking elements of the same type as x to elements of the type of e.

$$\frac{\mathcal{E} \vdash x : s, \mathcal{E} \vdash e : t}{\mathcal{E} \vdash f\ x = e : s \rightarrow t}$$

  - Essentially, it is necessary to maintain a **type environment** $\mathcal{E}$, and type inferences are done with respect to that environment.

Type Inference

- These rules impose constraints as follows. Suppose we want to do type inference on a given expression e. We first assign unique type variables $'a$:
  - one to each variable x occurring in e, and
  - one to each occurrence of each subexpression of e.
- In the 1^{st} clause, the type variable is associated with the variable (x), and in the 2^{nd}, it is associated with the occurrence of the subexpression in e. Call the type variable assigned to x in the 1^{st} clause $u(x)$, and call the type variable assigned to occurrence of a subexpression e in the 2^{nd} clause $v(e)$
- Now we take the following constraints:
  - $u(x) = v(x)$ for each occurrence of a variable x
  - $v(e_1) = v(e_2) \rightarrow v((e_1\ e_2))$ for each occurrence of a subexpression $(e_1\ e_2)$.
    That is, the type $v(e_1)$ of function e_1 has the type $v(e_2)$ of its parameter e_2 as domain and the type $v((e_1\ e_2))$ of its application as co-domain
  - $v(\mathit{fun}\ x = e) = v(x) \rightarrow v(e)$ for each occurrence of a subexpression $\mathit{fun}\ x = e$
- To infer the polymorphic type of an expression e, we walk the abstract syntax tree of e and collect these constraints, then perform unification on the constraints to obtain their mgu. The resulting substitution applied to the type variable $v(e)$ gives the most general polymorphic type of e.

Type Inference

- The complete code for unification and simple polymorphic type inference can be downloaded from here. It is written in ML.
- The following is some sample output from the inference system where '?' is the command prompt and the ticked variables (like 'a) are type variables. Note how the most general polymorphic type is inferred in different cases.

```
? fun x -> x
fun x -> x : 'a -> 'a
? fun x -> fun y -> x
fun x -> fun y -> x : 'a -> 'b -> 'a
? fun x -> fun y -> y
fun x -> fun y -> y : 'a -> 'b -> 'b
? fun f -> fun g -> fun x -> f (g x)
fun f -> fun g -> fun x -> f (g x) : ('e -> 'd) -> ('c -> 'e) -> 'c -> 'd
? fun x -> fun y -> fun z -> x z (y z)
fun x -> fun y -> fun z -> x z (y z) : ('c -> 'e -> 'd) -> ('c -> 'e) -> 'c -> 'd
? fun f -> (fun x -> f x x) (fun y -> f y y)
not unifiable: circularity
?
```

- The last term is not typable because unification results in a circularity. Recall the self-application in λ

$$sa = \lambda x.\ x\ x$$

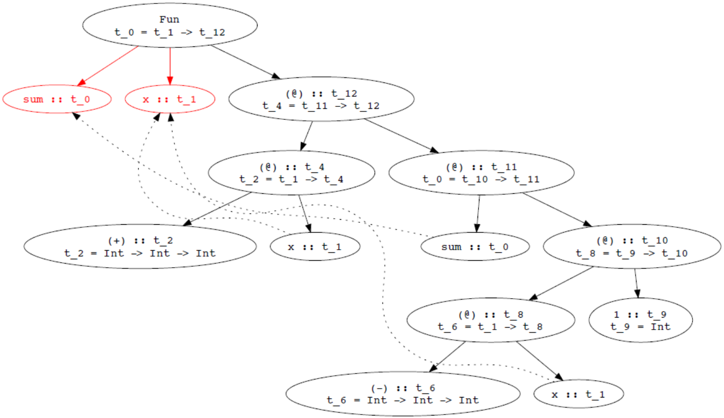
Type Inference: Examples

Sources:

- *Concepts in Programming Languages* by John C. Mitchell, Cambridge University Press, 2003

Type Inference Algorithm: Example 5: Recursive Function: sum

- `sum x = x + sum (x-1)`
  `> sum :: Int -> Int`



Parse Tree for sum Function Annotated with Type Variables and Associated Constraints

Type Inference Algorithm: Example 5: Recursive Function: `sum`

Add Constraints

```
t_0  = t_1 -> t_12
t_0  = t_10 -> t_11
t_2  = t_1 -> t_4
t_2  = Int -> Int -> Int
t_4  = t_11 -> t_12
t_6  = t_1 -> t_8
t_6  = Int -> Int -> Int
t_8  = t_9 -> t_10
t_9  = Int
```

Solve Constraints

```
t_0  = Int -> Int
t_1  = t_10 = Int
t_2  = Int -> (Int -> Int)
t_4  = Int -> Int
t_6  = Int -> (Int -> Int)
t_8  = Int -> Int
t_9  = Int
t_10 = t_1 = Int
t_11 = t_12 = Int
t_12 = t_11 = Int
```

- `sum x = x + sum (x-1)`
  `> sum :: Int -> Int`
- As the constraints can be solved, the function is *typeable*. By the solution of the constraints, the type of `sum` (the type `t_0`) is `Int -> Int`

Type Inference Algorithm: Example 6: Polymorphic Datatypes: `length`

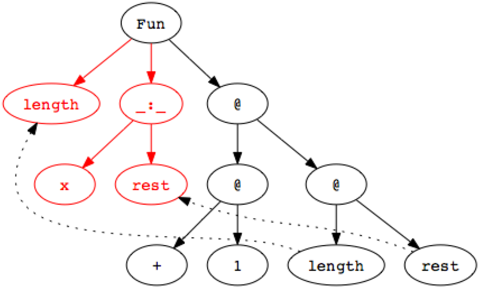
- Functions may have multiple clauses

```
length [] = 0
length (x:rest) = 1 + (length rest)
> length :: [t] -> Int
```

- Type inference
  - Infer separate type for each clause
  - Combine by adding constraint that all clauses must have the same type
  - Recursive calls: function has same type as its definition

Type Inference Algorithm: Example 6: Polymorphic Datatypes: `length`

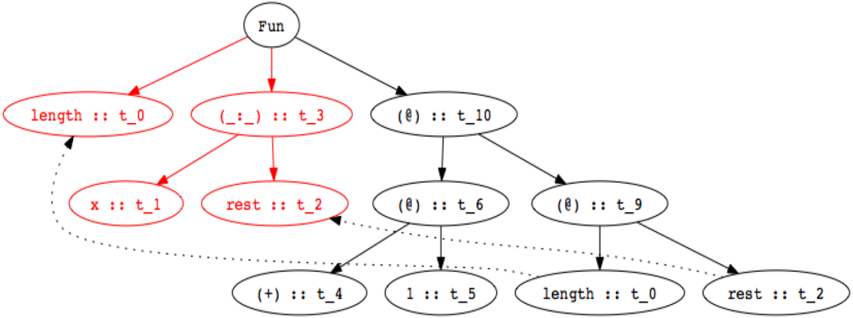
- `length (x:rest) = 1 + (length rest)`
- The `length` function has a type involving type variables, making the function polymorphic
- **Parse Program** text to construct parse tree



Parse Tree for `length` Function

Type Inference Algorithm: Example 6: Polymorphic Datatypes: `length`

- `length (x:rest) = 1 + (length rest)`
- **Assign type variables to nodes**



Parse Tree Labeled with Type Variables

Type Inference Algorithm: Example 6: Polymorphic Datatypes: `length`


```
length (x:rest) = 1 + (length rest)
```

Add Constraints

```
t_0 = t_3 -> t_10
t_3 = t_2
t_3 = [t_1]
t_6 = t_9 -> t_10
t_4 = t_5 -> t_6
t_4 = Int -> Int -> Int
t_5 = Int
t_0 = t_2 -> t_9
```

Conforms for:

```
length [] = 0
> length :: [t_1] -> Int
```

Solve Constraints

```
t_6 = Int -> Int
t_4 = Int -> Int -> Int
t_5 = Int
t_9 = Int
t_10 = Int
t_0 = [t_1] -> Int
```

Type Inference Algorithm: Example 6: Multi-Clause Function: append

- Type inference for functions with several clauses may be done by a type check of each clause separately. Then, because all clauses define the same function, we impose the constraint that the types of all clauses must be equal.
- ```
  append ([], r) = r
  append (x:xs, r) = x : append(xs, r)
  > append :: ([t], [t]) -> [t]
  ```
- As the type `([t], [t]) -> [t]` indicates, append can be applied to any pair of lists, as long as both lists contain the same type of list elements. Thus, append is a polymorphic function on lists.
  ```

Type Inference Algorithm: Example 6: Multi-Clause Function: append

- ```
  append ([], r) = r
  append (x:xs, r) = x : append(xs, r)
  > append :: ([t], [t]) -> [t]
  ```
- Intuitively, the first clause has type

  `([t], t_1) -> t_1`

  because the first argument must match the empty list `[]`, but the second argument may be anything
- The second clause has type

  `([t], t_1) -> [t]`

  because the return result is a list containing one element from the list passed as the first argument.
- If we require that the two clauses have the same type by imposing the constraint

  `([t], t_1) -> t_1 = ([t], t_1) -> [t]`

  we must have

  `t_1 = [t]`
- This equality gives us the final type for `append`:

  `append :: ([t], [t]) -> [t]`
  ```

Homework 1: `reverse`

Infer the types of the following:

[1]
```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse xs ++ [x]
```
where `x` (`xs`) is the head (tail) of the list, `[x]` is the list builder, `++` is the concatenation operator

[2]
```
appendreverse xs =
    let rev ( [], elem ) = elem
        rev ( y:ys, elem ) = rev( ys, y:elem)
    in rev( xs, [] )
```

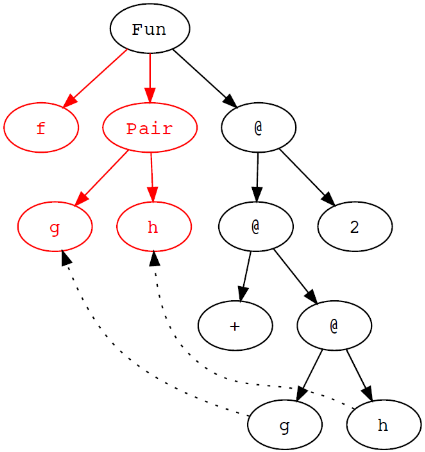
[3]
```
reverse2 xs  = app ([], xs)
       where
       app (ys, [])     = ys
       app (ys, (x:xs)) = app ((x:ys), xs)
```

[4]
```
reverseW [] = []
reverseW (x:xs) = reverseW xs
reverseW :: [t] -> [t]
```

Comment on the type correctness and logic correctness of the function `reverseW`

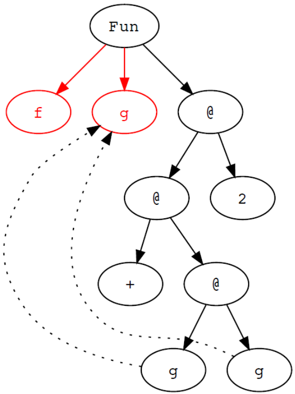
Homework 2: `apply`: modified

Use the below parse graph to calculate the μHaskell type for the function `f(g, h) = g(h) + 2`
Assume that `2` has type `Integer` and `+` has type `Integer -> Integer -> Integer`



Homework 3: `apply`: self-apply

Use the below parse graph to trace the Haskell type-inference algorithm on the function `f g = (g g) + 2` where `2` has type `Integer` and `+` has type `Integer -> Integer -> Integer`



What is the output of the type checker?

Type Deduction in C++03/C++11/C++14

Sources:

- *Concepts in Programming Languages* by John C. Mitchell, Cambridge University Press, 2003
- Ad-hoc, Inclusion, Parametric & Coercion Polymorphisms
- The Four Polymorphisms in C++, 2022

Type Deduction in C++

- C++ is a strongly typed languages
- Type System of C++ has evolved and is evolving with the the dialects of the language to provide:
  - Type Safety: C++03 onward
  - Generic Programming: C++03 onward; but much enhanced from C++11
  - Performance Optimization (with better semantics by type): C++11 onward
- In this section, we first discuss how type systems in C++ support polymorphisms of various kinds (originally framed in C++98 and later enhanced) and then elucidate various refinements in C++11 and C++14

Type Deduction in C++: Polymorphism

Polymorphism

- Polymorphism (or *having multiple forms*), refers to constructs taking different types by need
- There are four forms of polymorphism in languages like C++:
  - *Ad hoc polymorphism*: [**Overloading**: *Compile time*]: Two or more implementations with different types are referred to by the same name. [**Overriding**] is a variant for a hierarchy
    Ad-hoc Polymorphism allows functions having same name to act differently for different types
  - *Parametric / Early Binding polymorphism*: [**Template**: *Compile time*]: A function may be applied to any arguments whose types match a type expression involving type variables
    Parametric Polymorphism opens a way to use the same code for different types
  - *Subtype / Inclusion polymorphism*: [**Dynamic Dispatch**: *Run time*]: The subtype relation between types allows an expression to have many possible types. This is *Late Binding polymorphism*
    Inclusion Polymorphism is the ability to use derived classes through base class pointers and references
  - *Coercion polymorphism*: [**(Implicit / Explicit) Casting**: *Compile / Run time*]: One type of object is used for another
    Coersion Polymorphism occurs when an object or primitive is cast into some other type

Support for Polymorphism

- *Ad hoc polymorphism*: [**Overloading** / **Overriding**: *Compile time*]:
  - Overload resolution by types of parameters
  - Overriding resolution considers the type of invoking object's type of `this` pointer
  - We outline the strategy here
- *Parametric / Early Binding polymorphism*: [**Template**: *Compile time*]:
  - Template type deduction
  - We outline the basic strategy through function templates and then elaborate for C++11
- *Subtype / Inclusion polymorphism*: [**Dynamic Dispatch**: *Run time*]:
  - Virtual functions and Virtual Function Tables in classes
  - CRT or C++ Run-time system
- *Coercion polymorphism*: [**(Implicit / Explicit) Casting**: *Compile / Run time*]:
  - Cast operators and overload resolution

Type Deduction in C++: Polymorphism: Ad-hoc Polymorphism

Ad-hoc Polymorphism: Overloading: Overload Resolution

- To resolve overloaded functions with one parameter
  - [1] Identify the set of *Candidate Functions*
  - [2] From the set of candidate functions identify the set of *Viable Functions*
  - [3] Select the *Best viable function* through (*Order is important*)
    - Exact Match
    - Promotion
    - Standard type conversion
    - User defined type conversion

Ad-hoc Polymorphism: Overloading: Resolution: Candidate Function

Step 1: Find candidate functions via name lookup. Unqualified calls will perform both regular unqualified lookup as well as argument-dependent lookup (if applicable).

Sources: Steps of Overload Resolution, Overloaded Method Resolution, Function overload resolution

Ad-hoc Polymorphism: Overloading: Resolution: Viable Function

Step 2: Filter the set of candidate functions to a set of *viable functions*. A viable function for which there exists an implicit conversion sequence between the arguments the function is called with and the parameters the function takes.

```
void f(char);          // (1)
void f(int ) = delete; // (2)
void f();              // (3)
void f(int&);          // (4)

f(4); // 1,2 are viable (even though 2 is deleted!)
      // 3 is not viable because the argument lists don't match
      // 4 is not viable because we cannot bind a temporary to a non-const lvalue reference
```

Ad-hoc Polymorphism: Overloading: Resolution: Best Match Function

Step 3: Pick the best viable candidate. A viable function F1 is a better function than another viable function F2 if the implicit conversion sequence for each argument in F1 is not worse than the corresponding implicit conversion sequence in F2, and...:

Step 3.1: For some argument, the implicit conversion sequence for that argument in F1 is a better conversion sequence than for that argument in F2, or

```
void f(int );  // (1)
void f(char ); // (2)

f(4);  // call (1), better conversion sequence
```

Ad-hoc Polymorphism: Overloading: Resolution: Best Match Function

Step 3.2: In a user-defined conversion, the standard conversion sequence from the return of F1 to the destination type is a better conversion sequence than that of the return type of F2, or

```
struct A {
    operator int();
    operator double();
} a;

int i = a; // a.operator int() is better than a.operator double() and a conversion
float f = a; // ambiguous
```

Step 3.3: In a direct reference binding, F1 has the same kind of reference by F2 is not, or

```
struct A {
    operator X&();  // #1
    operator X&&(); // #2
};
A a;
X& lx = a;  // calls #1
X&& rx = a; // calls #2
```

Ad-hoc Polymorphism: Overloading: Resolution: Best Match Function

Step 3.4: F1 is not a function template specialization, but F2 is, or

```
template <class T> void f(T ); // #1
void f(int );                  // #2

f(42); // calls #2, the non-template
```

Step 3.5: F1 and F2 are both function template specializations, but F1 is more specialized than F2

```
template <class T> void f(T );  // #1
template <class T> void f(T* ); // #2

int* p;
f(p); // calls #2, more specialized
```

Ambiguity: If there's no single best viable candidate at the end, the call is ambiguous:

```
void f(double) { }
void f(float)  { }

f(42); // error: ambiguous
```

Overload Resolution: Exact Match

- lvalue-to-rvalue conversion
  - Most common
- Array-to-pointer conversion

  Definitions:
  ```
  int ar[10];
  void f(int *a);
  ```
  Call:
  ```
  f(ar)
  ```
- Function-to-pointer conversion

  Definitions:
  ```
  typedef int (*fp) (int);
  void f(int, fp);
  int g(int);
  ```
  Call:
  ```
  f(5, g)
  ```
- Qualification conversion
  - Converting pointer (only) to `const` pointer

Overload Resolution: Promotion & Conversion

- Examples of Promotion
  - `char` to `int`; `float` to `double`
  - `enum` to `int` / `short` / `unsigned int` / ...
  - `bool` to `int`
- Examples of Standard Conversion
  - integral conversion
  - floating point conversion
  - floating point to integral conversion

    The above 3 may be dangerous!
  - pointer conversion
  - bool conversion

Example: Overload Resolution with one parameter

- In the context of a list of function prototypes:

```
int g(double);                  // F1
void f();                       // F2
void f(int);                    // F3
double h(void);                 // F4
int g(char, int);               // F5
void f(double, double = 3.4);   // F6
void h(int, double);            // F7
void f(char, char *);           // F8
```

  The call site to resolve is:

```
f(5.6);
```

- Resolution:
  - Candidate functions (by name): F2, F3, F6, F8
  - Viable functions (by # of parameters): F3, F6
  - Best viable function (by type `double` – Exact Match): F6

Example: Overload Resolution fails

- Consider the overloaded function signatures:

```
int fun(float a) {...}             // Function 1
int fun(float a, int b) {...}      // Function 2
int fun(float x, int y = 5) {...} // Function 3

int main() {
    float p = 4.5, t = 10.5;
    int s = 30;

    fun(p, s); // CALL - 1
    fun(t);    // CALL - 2
    return 0;
}
```

- `CALL - 1`: Matches Function 2 & Function 3
- `CALL - 2`: Matches Function 1 & Function 3
- Results in ambiguity

Type Deduction in C++: Polymorphism: Parametric Polymorphism

Parametric Polymorphism

- Parametric polymorphism may be **implicit** or **explicit**
- In **explicit parametric polymorphism**, the program text contains type variables that determine the way that a function or other value may be treated polymorphically.
  - In addition, explicit polymorphism often involves **explicit instantiation** or type application to indicate how type variables are replaced with specific types in the use of a polymorphic value.
  - C++ templates are a well-known example of explicit polymorphism.
- Haskell polymorphism is called **implicit parametric polymorphism** because programs that declare and use polymorphic functions do not need to contain types – the type-inference algorithm computes when a function is polymorphic and computes the instantiation of type variables as needed.
  - *The main characteristic of parametric polymorphism is that the set of types associated with a function or other value is given by a type expression that contains type variables*. For example, a Haskell function that sorts lists might have the Haskell type

```
sort :: ((t, t) -> Bool, [t]) -> [t]
```

Parametric Polymorphism: C++ Function Templates

- To swap values of variables of *any* type, define function template using type variable T:

```
template <typename T>
void Swap(T& x, T& y) { T tmp = x; x = y; y = tmp; }
```

- `let Swap` = $\lambda(x : T).\lambda(y : T).\ E$, where T is the type variable
- Templates allow us to treat `Swap` as a function with a type argument
- In C++, function templates are instantiated automatically as needed. For example:

```
int i,j; ... Swap(i,j);    // replace T with int
float a,b; ... Swap(a,b);  // replace T with float
string s,t; ... Swap(s,t); // replace T with String
```

- Applying type inference for `int i,j; ... Swap(i,j);`:
  - By Abstraction
    ```
    Swap (x, y) = ...
    Swap :: (T&, T&) -> void
    ```
  - By Application
    ```
    int i,j; ... Swap(i,j);
    Swap :: (int , int) -> void
    ```
  - `T = int`

Parametric Polymorphism: C++ Function Templates

- ```
  template <typename T>
  void Swap(T& x, T& y) { T tmp = x; x = y; y = tmp; }

  class IntWrap {
      int data;
  }
  ```
- Examples:

```
int i,j;              Swap(i,j);              T = int
double c,d;           Swap(c,d);              T = double
string s,t;           Swap(s,t);              T = string
IntWrap a,b;          Swap(a,b);              T = IntWrap

int i,j;              Swap<int>(i,j);         T = int
double c,d;           Swap<double>(c,d);      T = double
string s,t;           Swap<string>(s,t);      T = string
IntWrap a,b;          Swap<IntWrap>(a,b);     T = IntWrap

const int ci, di;     Swap(ci,di);            T = const int    const cannot be assigned
int i, double d;      Swap(i,d);              T = ?            template parameter
                                                               'T' is ambiguous
int i, j;             Swap<double>(i,j);      T = double       cannot convert
                                                               argument 1 from 'int' to 'double
```
```

Parametric Polymorphism: C++ Function Templates

- ```
  template <typename T>
  void Swap(T& x, T& y) { T tmp = x; x = y; y = tmp; }

  class Uncopyable {
  protected:
      Uncopyable& operator=(const Uncopyable& u);
  };

  class IntWrap : public Uncopyable {
          int data;
  };
  ```
- Examples:

```
IntWrap a,b;   Swap(a,b);             T = IntWrap
               Swap<IntWrap>(a,b);
```

Link Error:
unresolved external symbol "
protected: class
Uncopyable & __thiscall
Uncopyable::operator=(class
Uncopyable const &)"
```

Parametric Polymorphism: C++ Function Templates

- ```
  template<typename T> T Add(T& a, T&b) { return a + b; }

  class IntWrap { int i;
  public: IntWrap(int i_) : i(i_) {}
      friend IntWrap operator+(IntWrap& a, IntWrap& b) { return IntWrap(a.i + b.i); }
  };
  ```
- `let Add =` $\lambda(a : T).\lambda(b : T).\ (a + b)$, where T is the type variable
- Examples:

`int i,j;`	`Add(i,j);`	`T = int`
`double c,d;`	`Add(c,d);`	`T = double`
`string s,t;`	`Add(s,t);`	`T = string`
`IntWrap a,b;`	`Add(a,b);`	`T = IntWrap`
`int i,j;`	`Add<int>(i,j);`	`T = int`
`double c,d;`	`Add<double>(c,d);`	`T = double`

Mixed Mode Addition

`int i, double d;`	`Add(i,d);`	`T = ?`	template parameter 'T' is ambiguous
`int i, double d;`	`Add(d,i);`	`T = ?`	template parameter 'T' is ambiguous
`int i, double d;`	`Add<int>(i,d);`	`T = int`	no instance of function template "Add" matches the arg. list arg. types are: (int, double)
`int i, double d;`	`Add<double>(i,d);`	`T = double`	no instance of function template "Add" matches the arg. list arg. types are: (double, int)
  ```

Parametric Polymorphism: C++ Function Templates

- ```
  template<typename T1, typename T2> T1 Add(T1& a, T2&b) { return a + b; }

  class IntWrap { int i;
  public: IntWrap(int i_) : i(i_) {}
      friend IntWrap operator+(IntWrap& a, IntWrap& b) { return IntWrap(a.i + b.i); }
  };
  ```
- `let Add =` $\lambda(a : T1).\lambda(b : T2).\ (a + b)$, where $T1$, and $T2$ are type variables
- Examples:

```
int i,j;          Add(i,j);   T1 = T2 = int
double c,d;       Add(c,d);   T1 = T2 = double
string s,t;       Add(s,t);   T1 = T2 = string
IntWrap a,b;      Add(a,b);   T1 = T2 = IntWrap
```

Mixed Mode Addition

```
int i, double d;  Add(i,d);   T1 = int, T2 = double    warning: 'return' : conversion
                                                        from 'double' to 'int'
int i, double d;  Add(d,i);   T1 = double, T2 = int
```
```

Parametric Polymorphism: C++ Function Templates


- ```
  template<typename T1, typename T2, typename R>
  R Add(T1& a, T2&b) { return a + b; }

  class IntWrap { int i;
  public: IntWrap(int i_) : i(i_) {}
      friend IntWrap operator+(IntWrap& a, IntWrap& b) { return IntWrap(a.i + b.i); }
  };
  ```
- `let Add` = $\lambda(a : T1).\lambda(b : T2).\lambda(r : R).\ (a + b)$, where $T1$, $T2$, and R are type variables
- Examples:

`int i,j;`	`Add(i,j);`	`T1 = T2 = int, R = ?`	could not deduce template argument for 'R'
`double c,d;`	`Add(c,d);`	`T1 = T2 = double, R = ?`	-do-
`string s,t;`	`Add(s,t);`	`T1 = T2 = string, R = ?`	-do-
`IntWrap a,b;`	`Add(a,b);`	`T1 = T2 = IntWrap, R = ?`	-do-

Mixed Mode Addition

`int i, double d;`	`Add(i,d);`	`T1 = int, T2 = double, R = ?`	-do-
`int i, double d;`	`Add(d,i);`	`T1 = double, T2 = int, R = ?`	-do-
  ```

Parametric Polymorphism: C++ Function Templates

- ```
  template<typename T1, typename T2, typename R>
  R Add(T1& a, T2&b) { return a + b; }

  class IntWrap { int i;
  public: IntWrap(int i_) : i(i_) {}
      friend IntWrap operator+(IntWrap& a, IntWrap& b) { return IntWrap(a.i + b.i); }
  };
  ```
- Examples:

```
int i,j;           Add<int, int, int>(i,j);              T1 = T2 = int. R = int
double c,d;        Add<double, double, double>(c,d);     T1 = T2 = double. R = double
string s,t;        Add<string, string, string>(s,t);     T1 = T2 = string. R = string
IntWrap a,b;       Add<IntWrap, IntWrap, IntWrap>(a,b);  T1 = T2 = IntWrap. R = IntWrap
```

Mixed Mode Addition

```
int i, double d;   Add<int, double, double>(i,d);        T1 = int, T2 = double
                                                         R = double
int i, double d;   Add<double, int, double>(d,i);        T1 = double, T2 = int
                                                         R = double
```
```

Parametric Polymorphism: C++ Function Templates

- Using Partial Template Specialization

```
template<typename T1, typename T2, typename R> // Fn-3
R Add(T1& a, T2& b) { return a + b; }

template<typename T1, typename T2> // Fn-2. Replace R by T1
T1 Add(T1& a, T2& b) { return Add<T1, T2, T1>(a, b); }

template<typename T> // Fn-1. Replace T1, T2, and R by T
T Add(T& a, T& b) { return Add<T, T, T>(a, b); }

template<> // Fn-0. Replace T1, T2, and R by int
int Add(int& a, int& b) { return Add<int, int, int>(a, b); }

class IntWrap { int i; public: IntWrap(int i_) : i(i_) {}
    friend IntWrap operator+(IntWrap& a, IntWrap& b) { return IntWrap(a.i + b.i); }
};
```

- Examples:

```
int i,j;            Add(i,j);    Fn-0    string s,t;    Add(s,t);    Fn-1
double c,d;         Add(c,d);    Fn-1    IntWrap a,b;   Add(a,b);    Fn-1
int i, double d;    Add(i,d);    Fn-2
int i, double d;    Add(d,i);    Fn-2
int i, double d;    Add<int, double, double>(i,d);                   Fn-3
int i, double d;    Add<double, int, double>(d,i);                   Fn-3
```

Type Deduction in C++11/C++14: auto & decltype

Sources:

- auto and decltype isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Placeholder type specifiers (since C++11) and decltype specifier, cppreference.com
- C++ auto and decltype Explained

auto

- In C++03, auto designated an object with *automatic storage type*. That is now *deprecated* We must specify the type of an object at declaration though the declaration may include an initializer with type
- In C++11 auto variables get the type from their *initializing expression*:

```
auto x1 = 10;              // x1: int

std::map<int, std::string> m;
auto i1 = m.begin();       // i1: std::map<int, std::string>::iterator
```

- const/volatile and reference/pointer adornments may be added:

```
const auto *x2 = &x1;      // x2: const int*
const auto& i2 = m;        // i2: const std::map<int, std::string>&
```

- To get a const_iterator, use cbegin (or cend, crbegin, and crend) container function:

```
auto ci = m.cbegin();      // ci: std::map<int, std::string>::const_iterator
```

- Type deduction for auto is akin to that for template parameters:

```
template<typename T> void f(T t);
f(expr);        // deduce T's type from expr
auto v = expr; // do essentially the same thing for v's type
```

auto

- For variables *not explicitly* declared to be a *reference*:
  - Top-level consts / volatiles in the initializing type are *ignored*
  - *Array and function names* in initializing types decay to *pointers*

```
const std::list<int> li;
auto v1 = li;               // v1: std::list<int>
auto& v2 = li;              // v2: const std::list<int>&

float data[BufSize];
auto v3 = data;             // v3: float*
auto& v4 = data;            // v4: float (&)[BufSize]
```

- Both *direct and copy initialization syntax* are permitted

```
auto v1(expr);              // direct initialization syntax
auto v2 = expr;             // copy initialization syntax
```

  For auto, both syntaxes have the same meaning
- auto is closely related to decltype and has extensive use in templates and generic lambdas

```
template<class T, class U> void multiply(const vector<T>& vt, const vector<U>& vu) {
    // ...
    auto tmp = vt[i]*vu[i]; // Compiler knows the type of tmp: product of T by a U
    // ...
}
```

decltype

- decltype yields the *type of an expression without evaluating it*

```
int x, *ptr;
decltype(x) i1;            // i1's type is int
decltype(ptr) p1;          // p1's type is int*
std::size_t sz = sizeof(decltype(ptr[44])); // sz = sizeof(int); ptr[44] not evaluated
```

- Fairly intuitive, but some quirks, for example, parentheses can matter:

```
struct S { double d; };
const S* p;
decltype(p->d) x1;                       // double
decltype((p->d)) x2;                     // const double&
```

  - Quirks rarely relevant (and can be looked up when necessary)

- Can simplify complex type expressions

```
void f(const vector<int>& a, vector<float>& b) {
    typedef decltype(a[0]*b[0]) Tmp; // Type deonted by int * float
    for (int i=0; i<b.size(); ++i) {
        Tmp* p = new Tmp(a[i]*b[i]);
        // ...
    }
    // ...
}
```

auto / decltype: Semantic Differences

- auto and decltype both infer types from expressions; but they semantically differ:

```
#include <iostream>

int main() {
    int a = 5;          // int
    int& b = a;         // int&
    const int c = 7;    // const int
    const int& d = c;   // const int&

    // auto never deduces adornments like cv-qualifer or reference
    auto a_auto = a; // int
    auto b_auto = b; // int
    auto c_auto = c; // int
    auto d_auto = d; // int

    // cv-qualifer or reference needs to be explicitly added
    auto& b_auto_ref = a;         // int&
    const auto c_auto_const = a; // const int

    // decltype deduces the complete type of the expression
    decltype(a) a_dt;     // int        // [C++14] decltype(auto) a_dt_auto = a; // int
    decltype(b) b_dt = b; // int&       // [C++14] decltype(auto) b_dt_auto = b; // int&
    decltype(c) c_dt = c; // const int  // [C++14] decltype(auto) c_dt_auto = c; // const int
    decltype(d) d_dt = d; // const int& // [C++14] decltype(auto) d_dt_auto = d; // const int&
}
```

auto / decltype: Determining compiler-deduced types in C++

- Compiler deduces types of expressions in various contexts:
  - In C++03, types are inferred for implicit conversions, templates, etc.
  - In C++11, in addition, types are inferred for `auto` and `decltype`
- **How can we know the type deduced by the compiler?**
  - In C++ type is inferred at compiler time[2] - no support to know the inferred type
  - Debug in an IDE and check the type. This is possible only if the program compiles
  - Use compiler errors: Errors shown for: [Programiz - C++ Online Compiler]
    - ▷ Incomplete template: [Determining types deduced by the compiler in C++]

```
template<typename T> class KnowType;

int arr[] = { 1, 2, 3 }; // int [3]
KnowType<decltype(arr)> arr_type; // error: aggregate 'KnowType<int [3]> arr_type'
                                  // has incomplete type and cannot be defined
```

    - ▷ Incomplete type: [Using 'auto' type deduction - how to find out what type the compiler deduced?]

```
int arr[] = { 1, 2, 3 }; // int [3]
decltype(arr)::_;                 // error: decltype evaluates to 'int [3]', which
                                  // is not a class or enumeration type
```

[2] C++ is statically typed (except for dynamic polymorphism where there is typeid support for type)

auto / decltype: Determining compiler-deduced types in C++

- We may also use typeid operator to know the type
  - Not a good idea as typeid is meant for dynamic type
  - The name of type retured by typeid is encoded

```
#include <bits/stdc++.h> // includes all standard library, but is not a standard header file of GNU C++
                         // DO NOT USE
using namespace std;

int main() {
    int x;                                   // int
    char y;                                  // char
    cout << typeid(x).name() << endl;        // i
    cout << typeid(y).name() << endl;        // c

    vector<int> vi;                          // std::vector<int>
    vector<double> vd;                       // std::vector<double>
    cout << typeid(vi).name() << endl;       // St6vectorIiSaIiEE
    cout << typeid(vd).name() << endl;       // St6vectorIdSaIdEE

    auto it = vi.begin();                    // std::vector<int>::iterator
    decltype(vi.cbegin()) cit = vi.cbegin(); // std::vector<int>::const_iterator
    cout << typeid(it).name() << endl;       // N9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEE
    cout << typeid(cit).name() << endl;      // N9__gnu_cxx17__normal_iteratorIPKiSt6vectorIiSaIiEEEE
}
```

Type Deduction in C++11/C++14: Suffix / Trailing Return Type

Sources:

- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Function declaration, cppreference.com
- When to use decltype(auto) versus auto?, cplusplus.com

Suffix / Trailing Return Type

- We really need `decltype` if we need a type for something *that is not a variable*, such as a *return type*. Consider:

```
template<class T, class U>
??? mul(T x, U y) { return x*y; }
```

- How to write the *return type*? It is the *type of* `x*y` – but how can we say that? Use `decltype`?

```
template<class T, class U>
decltype(x*y) mul(T x, U y) { return x*y; } // scope problem! types of x and y not known
```

- That won't work because `x` and `y` are not in scope. So:

```
template<class T, class U>
(decltype*(T*)(0)**(U*)(0)) mul(T x, U y) { return x*y; } // ugly! and error prone
```

- Put the return type where it belongs, after the arguments:

```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
```

We use the notation auto to mean *return type to be deduced or specified later*

Suffix / Trailing Return Type

- The *suffix syntax* is not primarily about *templates* and *type deduction*, it is really about *scope*

```
struct List {
    struct Link { /* ... */ };
    Link* erase(Link* p); // remove p and return the link before p
    // ...
};
List::Link* List::erase(Link* p) { /* ... */ }
```

- The first `List::` is necessary only because the scope of `List` is not entered until the second `List::`. Better:

```
auto List::erase(Link* p) -> Link* { /* ... */ } // No explicit qualification for Links
```

- To declare objects, `decltype` can replace `auto`, but more verbosely:

```
std::vector<std::string> vs;
auto i = vs.begin();
decltype(vs.begin()) i = vs.begin();
```

- Only `decltype` solves the template-return-type problem in C++11 (by Perfect Forwarding)
- `auto` is for everybody. `decltype` is primarily for template authors

Suffix / Trailing Return Type: C++14

- In C++11, we use suffix return type to specify return type of templates to be inferred:

```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
```

  This is unclean because the return expression has to be *repeated* within decltype

- In C++14, suffix type can be skipped and the return type is deduced directly:

```
template<class T, class U>
auto mul(T x, U y) { return x*y; }
```

  For compatibility, it still supports the suffix return type. Hence, the following is still valid:

```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
```

- C++14, further introduces decltype(auto) for deducing the return type by the semantics of decltype and not the semantics of auto. *No suffix return type is allowed here*

```
template<class T, class U>
decltype(auto) mul(T x, U y) { return x*y; }
```

- We present an example to highlight the differences between auto and decltype(auto)

Suffix / Trailing Return Type: C++14: auto / decltype(auto)

```
#include <iostream>

// returns prvalue: plain auto never deduces to a reference. prvalue is a pure rvalue - TBD later
template<typename T> auto foo(T& t) { return t.value(); }

// return lvalue: auto& always deduces to a reference
template<typename T> auto& bar(T& t) { return t.value(); }

// return prvalue if t.value() is an rvalue
// return lvalue if t.value() is an lvalue
// decltype(auto) has decltype semantics (without having to repeat the expression)
template<typename T> decltype(auto) foobar(T& t) { return t.value(); }

int main() {
    struct A { int i = 0 ; int& value() { return i ; } } a;
    struct B { int i = 0 ; int value() { return i ; } } b;

    foo(a) = 20; // *** error: expression evaluates to prvalue of type int
    foo(b);      // fine: expression evaluates to prvalue of type int

    bar(a) = 20; // fine: expression evaluates to lvalue of type int&
    bar(b);      // *** error: auto& always deduces to a reference (int&) - bar(b) needs an initializer

    foobar(a) = 20; // fine: expression evaluates to lvalue of type int&
    foobar(b);      // fine: expression evaluates to prvalue of type int
}
```

Suffix / Trailing Return Type: auto / declspec(auto)

- **Recommendations**
  - auto
    - Use auto to return a *prvalue with type deduction*
    - Use auto& or const auto& to return an *lvalue with type deduction*
  - declspec(auto)
    - Use declspec(auto) to write *forwarding templates*
    - Using decltype(auto), the return type is as what would be obtained if the expression used in the *return statement were wrapped in* decltype
    - Without decltype(auto), the deduction follows rules of *template argument deduction*
- **Summary**

```
auto foo() {                    // rt = int
    int x = 0; return (x); // returning local variable by value. Fine
}

decltype(auto) bar() {      // rt = int&
    int x = 0; return (x); // returning local variable by reference. Trouble
}
```

Source: When to use decltype(auto) versus auto?

Type Deduction in C++11/C++14: Copying vs. Moving

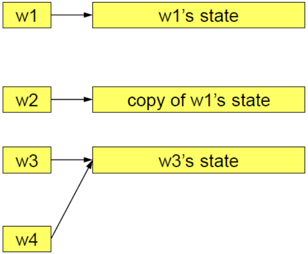
Sources:

- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++

Copying vs. Moving

- C++ has always supported copying object state:
  - *Copy* constructors, *Copy* assignment operators
- C++11 adds support for requests to *Move* object state:

```
Widget w1;
...
// copy w1's state to w2
Widget w2(w1);
Widget w3;
...
// move w3's state to w4
Widget w4(std::move(w3));
```

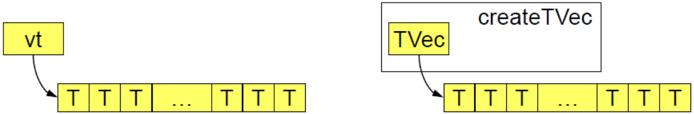


- **Note**: w3 continues to exist in a valid state after creation of w4

Copying vs. Moving: Return Value

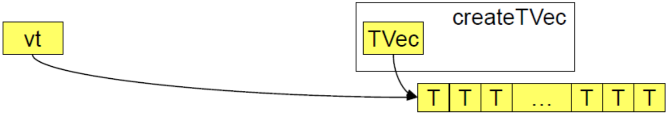
- C++ at times performs *extra copy*, while *temporary objects* are prime candidates for *move*:

```
typedef std::vector<T> TVec;
TVec createTVec(); // factory function
TVec vt;
...
vt = createTVec(); // in C++03, copy return value to vt, then destroy return value
```



- *Moving* values would be cheaper and C++11 generally turns such copy operations into moves:

```
TVec vt;
...
vt = createTVec(); // implicit move request in C++11. move data in return value object to vt
                   // then destroy return value object
```



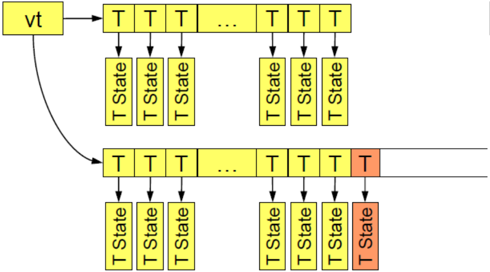
Copying vs. Moving: Append a Full Vector

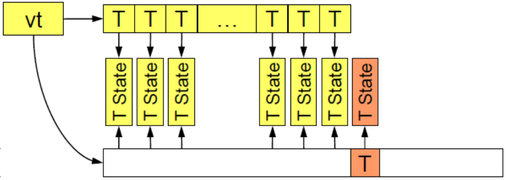
- Appending to a full vector causes much *copying before the append*. Moving would be efficient:

assume vt lacks unused capacity

```
std::vector<T> vt;
...
vt.push_back(T object);
```

```
std::vector<T> vt;
...
vt.push_back(T object);
```





- `vector` and `deque` operations like `insert`, `emplace`, `resize`, `erase`, etc. would benefit too

Copying vs. Moving: Swap

- Consider swapping two values:

By Copy

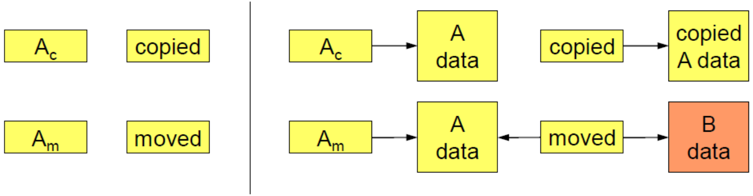
```
template<typename T> void swap(T& a, T& b) { // std::swap impl. by copy
void swap(T& a, T& b) {
    T tmp(a);           // copy a to tmp (=> 2 copies of a)
    a = b;              // copy b to a (=> 2 copies of b)
    b = tmp;            // copy tmp to b (=> 2 copies of tmp)
}                       // destroy tmp
```

By Move

```
template<typename T> void swap(T& a, T& b) { // std::swap impl. by move
    T tmp(std::move(a)); // move a's data to tmp
    a = std::move(b);    // move b's data to a
    b = std::move(tmp);  // move tmp's data to b
}                        // destroy (eviscerated) tmp
```

Copying vs. Moving: Deep vs. Shallow Copy

- Moving most important when:
  - *Object has data in separate memory (for example, on free store)*.
  - *Copying is deep*
- Moving copies only object memory
  - Copying copies object memory **+ separate memory**
- Consider copying/moving A to B:



- **Moving never slower than copying, and often faster**

Simple Performance Test

- Given

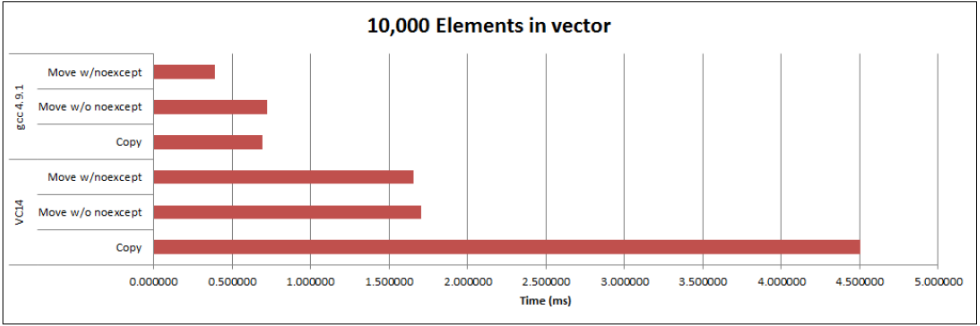
```
const std::string stringValue("This string has 29 characters");

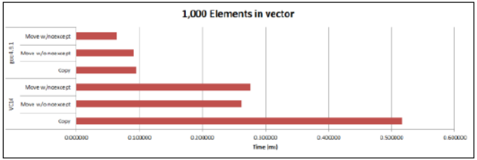
class Widget { std::string s;
public:
    Widget(): s(stringValue) { }
  ...
};
```

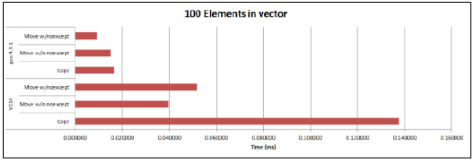
- Consider this push_back-based loop:

```
std::vector<Widget> vw;
Widget w;
for (std::size_t i = 0; i < n; ++i) {  // append n copies of w to vw
    vw.push_back(w);
}
```

Performance Data







Copying vs. Moving

- Lets C++ recognize move opportunities and take advantage of them.
  - **How recognize them?**
  - **How take advantage of them?**
- **Moving a key new C++11 idea**
  - Usually an optimization of copying
- Most standard types in C++11 are *move-enabled*
  - They support move requests
  - For example, STL containers
- Some types are *move-only*:
  - Copying prohibited, but moving is allowed
  - For example, stream objects, `std::thread objects`, `std::unique_ptr`, etc.

Sources:

- Rvalue references and move semantics, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Rvalue References
  - C++ Rvalue References Explained
  - Lvalues and Rvalues, accu.org, 2004
  - What are rvalues, lvalues, xvalues, glvalues, and prvalues?, stackoverflow.com, 2010
- Move Semantics
  - What is move semantics?, stackoverflow.com, 2010
  - M.3 — Move constructors and move assignment, learncpp.com, 2021
  - Move Constructors and Move Assignment Operators (C++), Microsoft, 2021
  - Understanding Move Semantics and Perfect Forwarding: Part 1, Part 2: Rvalue References and Move Semantics, Part 3: Perfect Forwarding, Drew Campbell, 2018

Type Deduction in C++11/C++14: Rvalue References and Move Semantics

Lvalues and Rvalues

- **Lvalues** *are generally things we can take the address of*:
  - In C, Expressions on *left-hand-side (LHS)* of an assignment
  - Named objects - variables
  - Legal to apply address of (&) operator
  - Lvalue references
- **Rvalues** *are generally things we cannot take the address of*:
  - In C, Expressions on *right-hand-side (RHS)* of an assignment
  - Typically unnamed temporary objects - expressions, return values from functions, etc.
  - Rvalue references
- Examples:

```
int x, *pInt;                   // x, pInt, *pInt are lvalues
std::size_t f(std::string str); // f and str are lvalues, f's return is rvalue
f("Hello");                     // temp string("Hello") created for call is rvalue
std::vector<int> vi;            // vi is lvalue
...
vi[5] = 0;                      // vi[5] is lvalue
```

  - Recall that `vector<T>::operator[]` returns `T&`

Moving and Lvalues

- *Value movement generally not safe when the source is an lvalue object*
  - That continues to exist, may be referred to later:

```
TVec vt1;
...
TVec vt2(vt1);      // it is expected that vt1 be copied to vt2, not moved!
...use vt1...       // value of vt1 here should be same as above
```

- *Value movement is safe when the source is an rvalue object*
  - Temp's usually *go away at statement's end*. No way to tell if their *value has been modified*

```
TVec createTVec();                  // as before
TVec vt1;
vt1 = createTVec();                 // rvalue source: move okay
auto vt2 = createTVec();            // rvalue source: move okay
vt1 = vt2;                          // lvalue source: copy needed
auto vt3(vt2);                      // lvalue source: copy needed
std::size_t f(std::string str);     // as before
f("Hello");                         // rvalue (temp) source: move okay
std::string s("C++11");
f(s);                               // lvalue source: copy needed
```

Rvalue References

- C++11 introduces **rvalue references**
  - Syntax: `T&&`
  - **Normal references** now known as **lvalue references**
    - ▷ Rvalue references behave similarly to Lvalue references
  - Must be initialized, cannot be rebound, etc.
- **Rvalue references identify objects that may be moved from**
- Reference Binding Rules
  - Important for overloading resolution
  - As always:
    - ▷ Lvalues may bind to *lvalue references*
    - ▷ Rvalues may bind to *lvalue references to* `const`
  - In addition:
    - ▷ Rvalues may bind to *rvalue references to non-*`const`
    - ▷ Lvalues may *not* bind to *rvalue references*
      - – Otherwise lvalues could be accidentally modified

Rvalue References

- Examples:

```
void f1(const TVec&);        // takes const lvalue ref
TVec vt;
f1(vt);                      // fine (as always)
f1(createTVec());            // fine (as always)
void f2(const TVec&);        // #1: takes const lvalue ref
void f2(TVec&&);             // #2: takes non-const rvalue ref
f2(vt);                      // lvalue => #1
f2(createTVec());            // both viable, non-const rvalue => #2
void f3(const TVec&&);       // #1: takes const rvalue ref
void f3(TVec&&);             // #2: takes non-const rvalue ref
f3(vt);                      // error! lvalue
f3(createTVec());            // both viable, non-const rvalue => #2
```

Rvalue References and const

- C++ remains const-correct:
  - const lvalues / rvalues bind only to references-to-const
- But rvalue-references-to-const are essentially useless
  - Rvalue references designed for two specific problems:
    - **Move semantics**
    - **Perfect forwarding**
  - C++11 language rules carefully crafted for these needs
    - Rvalue-refs-to-const not considered in these rules
  - const T&&s are legal, but not designed to be useful
    - Uses already emerging :-)
- Implications:
  - Do not declare const T&& parameters
  - Not possible to move from them, anyway
  - Hence this rarely makes sense:

Distinguishing Copying from Moving

- Overloading exposes move-instead-of-copy opportunities:

```
class Widget { public:
    Widget(const Widget&);                  // copy constructor
    Widget(Widget&&) noexcept;              // move constructor
    Widget& operator=(const Widget&);       // copy assignment op
    Widget& operator=(Widget&&) noexcept;   // move assignment op
  ...
};
Widget createWidget(); // factory function
Widget w1;
Widget w2 = w1;        // lvalue src => copy required
w2 = createWidget();   // rvalue src => move okay
w1 = w2;               // lvalue src => copy required
```

- Move operations need not be `noexcept`, but it is preferable
  - Moves should be fast, and `noexcept => more` optimizable
  - Some contexts require `noexcept` moves (for example, `std::vector::push_back`)
  - Move operations often have natural `noexcept` implementations
- We declare move operations `noexcept` by default

Copy vs. Move : Lvalue vs. Rvalue

```
class A { public: A() { std::cout << "Defa Ctor" << endl; }           // Defa Constructor
    A(const A&) { std::cout << "Copy Ctor" << endl; }                 // Copy Constructor
    A(A&&) noexcept { std::cout << "Move Ctor" << endl; }             // Move Constructor
    A& operator=(const A&) { cout << "Copy =" << endl; return *this; }     // Copy =
    A& operator=(A&&) noexcept { cout << "Move =" << endl; return *this; } // Move =
    friend A operator+(const A& a, const A& b) { A t; return t; } // Temp. obj. ret.-by-value
};
```

	Only Copy		Copy & Move	
	Debug	*Release*	*Debug*	*Release*
`A a; // lvalue`	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
`A b = a; // lvalue`	Copy Ctor	Copy Ctor	Copy Ctor	Copy Ctor
`A c = a + b; // rvalue`	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
`// RVO in a + b for release build`	Copy Ctor	// RVO	Move Ctor	// RVO
`A d = std::move(a); // rvalue`	Copy Ctor	Copy Ctor	Move Ctor	Move Ctor
`b = a; // lvalue`	Copy =	Copy =	Copy =	Copy =
`c = a + b; // rvalue`	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
	Copy Ctor	// RVO	Move Ctor	// RVO
	Copy =	Copy =	Move =	Move =

- Return Value Optimization (RVO) eliminates the temp. obj. created to hold a function's return value
- `std::move(t)` produces a rvalue from `t` to indicate that the object `t` may be *moved from*

Copy vs. Move : Lvalue vs. Rvalue: Explanation

```
class A { public: A() { std::cout << "Defa Ctor" << endl; }           // Defa Constructor
    A(const A&) { std::cout << "Copy Ctor" << endl; }                 // Copy Constructor
    A(A&&) noexcept { std::cout << "Move Ctor" << endl; }             // Move Constructor
    A& operator=(const A&) { cout << "Copy =" << endl; return *this; }     // Copy =
    A& operator=(A&&) noexcept { cout << "Move =" << endl; return *this; } // Move =
    friend A operator+(const A& a, const A& b) { A t; return t; } // Temp. obj. ret.-by-value
};
```

- `A a;` ⇒ `a` is an lvalue and is *default constructed*
- `A b = a;` ⇒ `a` is an lvalue and hence, `b` is *copy constructed*
- `A c = a + b;` ⇒ `operator+(a, b)` *default constructs* `t`, computes the result of `a + b` in `t` (not shown) and then *returns* `t` *by value*. Hence `a + b` is an rvalue and `c` is *move constructed* (if available, else *copy constructed*). **Note that in release (optimized) compiler build, RVO[3] allows `t` to be constructed directly in `c` and no copy or move construction is needed**
- `A d = std::move(a);` ⇒ `std::move` (in `<utility>`) can force an rvalue type. It produces an rvalue from `t` to indicate that the object `t` may be *moved from*. Hence `d` is *move constructed* (if available, else *copy constructed*)
- `b = a;` ⇒ `a` is an lvalue and hence, `b` is *copy assigned*
- `c = a + b;` ⇒ As above, `c` is *move assigned* (if available, else *copy assigned*) after *move construction* (if available, else *copy construction*). *Copy (move) construction* is eliminated by RVO in release build

[3] Return Value Optimization (RVO) eliminates the temp. obj. created to hold a function's return value

Copy vs. Move : Vector


```
// C++ program with the copy and the move constructors

class C { int* data; // Declare the raw pointer as the data member of class
public:
    C(int d) {                // Constructor
        data = new int(d); // Declare object in the heap
        cout << "Ctor: " << d << endl;
    };
    C(const C& src) : myClass{ *src.data } { // Copy Constructor by delegation
        // Copying the data by making deep copy
        cout << "C-Ctor: " << *src.data << endl;
    }
    C(C&& src) : data{ src.data } noexcept { // Move Constructor
        cout << "M-Ctor: " << *src.data << endl;
        src.data = nullptr;
    }
    ~C() { // Destructor
        if (data != nullptr) // If pointer is not pointing to nullptr
            cout << "Dtor: " << *data << endl;
        else                 // If pointer is pointing to nullptr
            cout << "Dtor: " << "nullptr " << endl;
        delete data; // Free up the memory assigned to the data member of the object
    }
};
```

Copy vs. Move : Vector


```
int main() { vector<C> v; // Create vector of C Class
    v.push_back(C{10});   // Inserting object of C class
    v.push_back(C{20});
}
```

	Only Copy			Copy & Move		
	Debug	Release	Remark	Debug	Release	Remark
`{ vector<C> v;`						
`// v.size() = 0`	Ctor: 10	Ctor: 10		Ctor: 10	Ctor: 10	
`v.push_back(C{10});`	Ctor: 10	Ctor: 10	// Delegate			
`// v.size() = 1`	C-Ctor: 10	C-Ctor: 10	// C-Ctor	M-Ctor: 10	M-Ctor: 10	// Add 10 to v
	Dtor: 10	Dtor: 10		Dtor: nullptr	Dtor: nullptr	
	Ctor: 20	Ctor: 20		Ctor: 20	Ctor: 20	
`// Move C{10}`	Ctor: 10	Ctor: 10	// Delegate			
`// for C{20}`	C-Ctor: 10	C-Ctor: 10	// C-Ctor	M-Ctor: 10	M-Ctor: 10	// Move 10 in v
`v.push_back(C{20});`	Dtor: 10	Dtor: 10		Dtor: nullptr	Dtor: nullptr	
`// v.size() = 2`	Ctor: 20	Ctor: 20	// Delegate			
	C-Ctor: 20	C-Ctor: 20	// C-Ctor	M-Ctor: 20	M-Ctor: 20	// Add 20 to v
	Dtor: 20	Dtor: 20		Dtor: nullptr	Dtor: nullptr	
`// End of scope`	Dtor: 10	Dtor: 10	// Release	Dtor: 10	Dtor: 10	// Release
`} // Release v`	Dtor: 20	Dtor: 20	// Vector v	Dtor: 20	Dtor: 20	// Vector v

Copy vs. Move : Vector: Explanation


```
class C { int* data; /* raw pointer */ public:
C(int d); /*Ctor*/ C(const C& src); /*C-Ctor*/ C(C&& src); /*M-Ctor*/ ~C(); /*Dtor*/ };
```

- `{ vector<C> v;` ⇒ `v` is *default constructed* as an empty vector of `C`. `v.size() = 0`
- `v.push_back(C{10});` ⇒ Construct `C{10}`, copy/move & place in `v[0]`, and destruct. `v.size() = 1`

```
Ctor: 10    /* Ctor for C{10} => t10, Temp.obj. and rvalue          */ Ctor: 10
Ctor: 10    /* delegated from C-Ctor                                */
C-Ctor: 10  /* C-Ctor for t10 => v10 = v[0], lvalue to place in v   */ M-Ctor: 10
Dtor: 10    /* Dtor for t10                                         */ Dtor: nullptr
```

- `v.push_back(C{20});` ⇒ Construct `C{20}`. Copy/move `v[0]` and destruct old `v[0]`. Copy/move & place `C{20}` in `v[1]`, and destruct. `v.size() = 2`

```
Ctor: 20    /* Ctor for C{20} => t20, Temp.obj. & rvalue            */ Ctor: 20
Ctor: 10    /* delegated from C-Ctor                                */
C-Ctor: 10  /* C-Ctor for v10 => v10_1 = v[0], lvalue to place in v */ M-Ctor: 10
Dtor: 10    /* Dtor for v10                                         */ Dtor: nullptr
Ctor: 20    /* delegated from C-Ctor                                */
C-Ctor: 20  /* C-Ctor for t20 => v20 = v[1], lvalue to place in v   */ M-Ctor: 20
Dtor: 20    /* Dtor for t20                                         */ Dtor: nullptr
```

- `}` ⇒ Automatic `v` going out of scope. Destruct `v[0]` and `v[1]`

```
Dtor: 10    /* Dtor for v10_1 = v[0]                                */ Dtor: 10
Dtor: 20    /* Dtor for v20 = v[1]                                  */ Dtor: 20
```

Copy vs. Move : Vector: Performance Trade-off

- Since, class `C` has no default constructor, `vector<C> v` is constructed as an empty vector with `v.size() = 0`. Hence, every time a `push_back` (`insert` at the `end()`) is done, we need to expand the allocation of the vector by copying / moving the existing elements
- For `v.push_back(C{10})`, `C{10}` is constructed as a temporary object (rvalue). So, it needs to be copied / moved for `push_back` to the vector as lvalue. Same for `v.push_back(C{20})`
- Further, for `v.push_back(C{20})`, fresh allocation and copy / movement of existing element is needed for `push_back`
- To `push_back` the n^{th} element, we need to copy / move existing $n-1$ elements. This means:
  - **Using Copy**
    - $n-1$ resource allocations (new int) and de-allocations (delete)
    - For n elements this adds to $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$ total allocations / de-allocations
  - **Using Move**
    - 0 resource allocations (new int) and de-allocations (delete)
    - For n elements this adds to $\sum_{i=0}^{n-1} 0 = 0$ total allocations / de-allocations. **Huge Benefit!**

Type Deduction in C++11/C++14: Implementing Move Semantics

Sources:

- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++

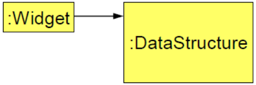
Implementing Move Semantics

- Move operations take source's value, but leave source in valid state:

```
class Widget {
public:
    Widget(Widget&& rhs) noexcept : pds(rhs.pds) // take source's value
        { rhs.pds = nullptr; }                   // leave source in valid state

    Widget& operator=(Widget&& rhs) noexcept {
        delete pds;         // get rid of current value
        pds = rhs.pds;      // take source's value
        rhs.pds = nullptr; // leave source in valid state
        return *this;
    }
    ...

private:
    struct DataStructure;
    DataStructure *pds;
};
```



- Easy for built-in types (for example, pointers). Trickier for UDTs...

Implementing Move Semantics

- Widget's move operator= fails given move-to-self:

```
Widget w;
w = std::move(w); // undefined behavior!
```

- It may be harder to recognize, of course:

```
Widget *pw1, *pw2;
...
*pw1 = std::move(*pw2); // undefined if pw1 == pw2
```

- C++11 condones this
  - In contrast to copy operator=
- A fix is simple, if you are inclined to implement it:

```
Widget& Widget::operator=(Widget&& rhs) noexcept {
    if (this == &rhs) return *this; // or assert(this != &rhs);
    ...
}
```

Implementing Move Semantics

- Part of C++11's `string` type:

```
string::string(const string&);       // copy constructor
string::string(string&&) noexcept;   // move constructor
```

- An incorrect move constructor:

```
class Widget { std::string s;
public:
    Widget(Widget&& rhs) noexcept // move constructor
        : s(rhs.s) // compiles, but copies!
    { ... }
    ...
};
```

- `rhs.s` an lvalue, because it has a name
  - Lvalueness / Rvalueness orthogonal to type!
    - `int`s can be lvalues or rvalues, and rvalue references can, too.
  - `s` initialized by `string`'s *copy* constructor

Implementing Move Semantics

- Another example:

```
class WidgetBase { public:
    WidgetBase(const WidgetBase&);      // copy ctor
    WidgetBase(WidgetBase&&) noexcept; // move ctor
    ...
};

class Widget: public WidgetBase { public:
    Widget(Widget&& rhs) noexcept      // move ctor
    : WidgetBase(rhs)                  // copies!
    { ... }
    ...
};
```

- rhs is an lvalue, because it has a name
  - Its declaration as Widget&& not relevant!

Move Semantics: How to code?

- **How to solve the problems in the coding of move semantics for UDTs?**
- In general, in a UDT, there are two kinds of constituent objects:
  - Data Member like `c.mRrc` `// MyClass c;`
  - Base Class part like `(MyClassBase&)c` `// MyClass: public MyClassBase`
- While we need to copy or move, we use the constructor or assignment operators of the underlying classes:
  - Construction: `mRrc(c.mRrc)` and `MyClassBase(c)`
  - Assignment: `mRrc = c.mRrc` and `MyClassBase::operator=(c)`
- For copy, we use the copy constructor / assignment operator
- For move, we need to use the move constructor / assignment operator to optimize resource handling
  - This means for the above four instances of the respective operators, the sources (`c.mRrc` or `c`) must be rvalues. But they are available by name as lvalues
- Hence, **we need a mechanism to convert an lvalue to an rvalue**
- `std::move` in `<utility>` provides for this

Type Deduction in C++11/C++14: std::move

Sources:

- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++
- Quick Q: What's the difference between std::move and std::forward?, isocpp.org
- On the Superfluousness of std::move – Scott Meyers, isocpp.org, 2012
- std::move, cppreference.com

Explicit Move Requests

- To request a move on an lvalue, use `std::move` from `<utility>`:

```
class MyClassBase { ... };
class MyClass: public MyClassBase { public:
    MyClass(MyClass&& c) noexcept : // Move Constructor
        MyClassBase(std::move(c)), // Request Move for base class part
        mRrc(std::move(c.mRrc))    // Request Move for data member
    { ... }
    MyClass& operator=(MyClass&& c) noexcept { // Move Assignment
        if (this != &c) {
            MyClassBase::operator=(std::move(c)); // Request Move for base class part
            mRrc = std::move(c.mRrc);              // Request Move for data member
        }
        return *this;
    }
    ...
};
```

- `std::move` turns lvalues into rvalues
  - The overloading rules do the rest

Explicit Move Requests

- `std::move` uses *implicit type deduction*
  Consider:

```
template<typename It>
void someAlgorithm(It begin, It end) {
    // permit move from *begin to temp

    // static_cast version
    auto temp1 =  static_cast<typename std::iterator_traits<It>::value_type&&>(*begin);

    // C-style cast version
    auto temp2 = (typename std::iterator_traits<It>::value_type&&)*begin;

    // std::move version
    auto temp3 = std::move(*begin);
    ...
```

- Great convenience by using `std::move`

Reference Collapsing in Templates

- In C++03, given

```
template<typename T> void f(T& param);
int x;
f<int&>(x);                          // T is int&
```

- f is initially instantiated as

```
void f(int& & param);                // reference to reference
```

- C++03's reference-collapsing rule says
  - `T& & => T&`
- So, after reference collapsing, f's instantiation is actually: `void f(int& param);`
- C++11's rules take rvalue references into account:
  - `T& & => T&     // from C++03`
  - `T&& & => T&    // new for C++11`
  - `T& && => T&    // new for C++11`
  - `T&& && => T&& // new for C++11`
- Summary:
  - *Reference collapsing involving a* `&` *is always* `T&`
  - *Reference collapsing involving only* `&&` *is* `T&&`

std::move: Return Type

- To guarantee an rvalue return type, std::move does this:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(MagicReferenceType obj) noexcept {
    return obj;
}
```

  - Recall that a T& return type would be an lvalue!
- Hence:

```
int x;
std::move<int&>(x); // calls remove_reference<int&>::type&& std::move(/*...*/)
                    // => int&& std::move(/*...*/)
```

  - Without std::remove_reference, move<int&> would return int&

std::move: Parameter Type

- Must be a non-const reference, because we want to move its value
- An lvalue reference does not work, because rvalues cannot bind to them:

```
TVec createTVec();                         // as before
TVec&& std::move(TVec& obj) noexcept;      // possible move instantiation
std::move(createTVec());                   // error!
```

- An rvalue reference does not, either, as lvalues cannot bind to them:

```
TVec&& std::move(TVec&& obj) noexcept; // possible move instantiation
TVec vt;
std::move(vt);                         // error!
```

- What std::move needs:
  - For lvalue arguments, a parameter type of T&
  - For rvalue arguments, a parameter type of T&&

std::move: Parameter Type: Solution by Overloading?

- Overloading could solve the problem:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T& lvalue) noexcept {
    return static_cast<std::remove_reference<T>::type&&>(lvalue);
}
template<typename T>
typename std::remove_reference<T>::type&&
move(T&& rvalue) noexcept {
    return static_cast<std::remove_reference<T>::type&&>(rvalue);
}
```

- But the *perfect forwarding problem*[4] would remain:
  - To forward n arguments to another function we would need 2^n overloads!
- *Rvalue references* aimed at both `std::move` and *perfect forwarding*

[4] *Perfect Forwarding Problem will be discussed in a later module*

std::move: T&& Parameter Deduction in Templates

- Given

```
template<typename T> void f(T&& param); // note non-const rvalue reference
```

- T's deduced type depends on what is passed to param:
  - **Lvalue** ⇒ T is an lvalue reference (T&)
  - **Rvalue** ⇒ T is a non-reference (T)
- In conjunction with reference collapsing:

```
int x;
f(x);              // lvalue: generates f<int&>(int& &&), calls f<int&>(int&)
f(10);             // rvalue: generates/calls f<int>(int&&)

TVec vt;           // typedef vector<int> TVec;
                   // TVec createTVec();
f(vt);             // lvalue: generates f<TVec&>(TVec& &&), calls f<TVec&>(TVec&)
f(createTVec());   // rvalue: generates/calls f<TVec>(TVec&&)
```

std::move: Implementation

- `std::move`'s parameter is thus `T&&`:

```
template<typename T>
typename std::remove_reference<T>::type&&
    move(T&& obj) noexcept {
        return obj;
    }
```

- This is almost correct. Problem:
  - `obj` is an lvalue (It has a name)
  - `move`'s return type is an rvalue reference
  - Lvalues cannot bind to rvalue references
- A cast eliminates the problem to give a correct implementation

```
template<typename T>
typename std::remove_reference<T>::type&&
    move(T&& obj) noexcept {
        using ReturnType = typename std::remove_reference<T>::type&&;
        return static_cast<ReturnType>(obj);
    }
```

T&& Parameters in Templates

- Compare conceptual and actual std::move declarations:

```
template<typename T>                        // conceptual
T&&
move(MagicReferenceType obj) noexcept;

template<typename T>                        // actual
typename std::remove_reference<T>::type&&
move(T&& obj) noexcept;
```

- T&& really is a magic[5] reference type!
  - For lvalue arguments, T&& becomes T& => lvalues can bind
  - For rvalue arguments, T&& remains T&& => rvalues can bind
  - For const/volatile arguments, const/volatile becomes part of T
  - T&& parameters can bind *anything*

[5] *Universal Reference will be discussed in a later module*

Move Semantics Project

Move Semantics Project

- To put the pieces of the Move Semantics puzzle together, we present a complete project with classes having resources of POD[6] and UDT
- We also code a resource management helper class (`ResMgr`) to *track the objects (resources) created and released dynamically*. This will help us to *quantify the benefits of move over copy*
- All functions are tracked with messages to understand *object lifetimes under copy and move*
- Using `ResMgr`, we present applications to compare the performance of the *Copy & Move* version of the classes against the *Copy Only* version
- The classes are (skeletons of `MyResource` and `MyClass` used earlier in the module):
  - `ResMgr`: *Resource Management Helper Class*. It has static member functions to `Create()`, `Release()` resources and print statistics (`Stat()`)
  - `MyResource`: *Resource Class with POD resource (`char*`)*. It has usual class members, overloaded output operator and a global function to call and return by value
  - `MyClass`: *Resource Class with UDT resource (`MyResource`)*. It has usual class members, overloaded output operator and a global function to call and return by value
- **The codes may be used to code move semantics in any project you develop**

[6] Plain Old Data (POD) refers to built-in types

Move Semantics: ResMgr: Resource Management Helper Class

```
#include <iostream>
#include <cstring>
using namespace std;

class ResMgr { // Resource Management class to track creation and release of resources
static unsigned int nCreated, nReleased; // Counters for created & released resources
public:
    ResMgr() { } // Constructor to be called before main
    ~ResMgr() {  // Destructor to be called after main
        cout << "\n\nResources Created = " << nCreated << endl;
        cout << "Resources Released = " << nReleased << endl;
    }
    inline static char *Create(const char *s) // Create a resource from s
    { return (s) ? ++nCreated, strdup(s) : nullptr; } // If s is not null, copy & increment counter
    inline static void Release(char *s) // Release the resource held by s
    { (s) ? free(s), ++nReleased : 0; } // If s is not null, increment counter & free resource
    inline static void Stat() // Print stats for resources created and released
    { cout << " Stat = (" << nCreated << ", " << nReleased << ")\n\n"; }
};
unsigned int ResMgr::nCreated = 0; // Define and initialize Counters
unsigned int ResMgr::nReleased = 0;

ResMgr m; // Static resource manager instance
          // Created before call to main(). Destroyed after return from main()
```

Move Semantics: MyResource: POD Resource Class - *Copy Only*


```
class MyResource { // Representative resource class with copy-only support
    char *str = nullptr; // Resource pointer
public:
    MyResource(const char* s = nullptr) : str(ResMgr::Create(s)) // Param. & Defa. Ctor
    { cout << "Ctor[R] "; } // Creates resource
    MyResource(const MyResource& s) : str(ResMgr::Create(s.str)) // Copy Ctor
    { cout << "C-Ctor[R] "; } // Copy-Creates resource
    MyResource& operator=(const MyResource& s) { cout << "C=[R] "; // Copy Assignment
        if (this != &s) { ResMgr::Release(str); str = ResMgr::Create(s.str); }
        return *this; // Releases and Copy-Creates resource
    }
    ~MyResource() // Destructor
    { cout << "Dtor[R] "; ResMgr::Release(str); } // Releases resource
    friend ostream& operator<<(ostream& os, const MyResource& s) { // Streams resource value
        cout << ((s.str) ? s.str : "null"); return os; // Streams "null" for nullptr (no resource)
    }
};

MyResource f(MyResource s)      // Global function
{ cout << "f[R] "; return s; } // Uses call-by-value & return-by-value
```

Move Semantics: Application using *Copy Only* MyResource


```
// ResMgr m is constructed here
int main() { // Appl. to check resource behavior for copy-only support through copy ctor & assignment
    MyResource r1{ "ppd" }; // Ctor[R] r1=ppd Stat = (1, 0)
    cout << "r1=" << r1; ResMgr::Stat(); // r1 constructed with parameter

    MyResource r2{ r1 }; // C-Ctor[R] r2=ppd r1=ppd Stat = (2, 0)
    cout << "r2=" << r2 << " r1=" << r1; ResMgr::Stat(); // r2 copy constructed from r1

    MyResource r3{ f(r2) }; // C-Ctor[R] f[R] C-Ctor[R] Dtor[R] r3=ppd r2=ppd Stat = (4, 1)
    cout << "r3=" << r3 << " r2=" << r2; ResMgr::Stat(); // r3 C-Ctor from f(r2): C-Ctor / Dtor for param

    r1 = r2; // C=[R] r1=ppd r2=ppd Stat = (5, 2)
    cout << "r1=" << r1 << " r2=" << r2; ResMgr::Stat(); // r1 copy assigned from r2

    MyResource r4; // Ctor[R] r4=null Stat = (5, 2)
    cout << "r4=" << r4; ResMgr::Stat(); // r4 default constructed

    r4 = f(r3); // C-Ctor[R] f[R] C-Ctor[R] C=[R] Dtor[R] Dtor[R] r4=ppd r3=ppd Stat = (8, 4)
    cout << "r4=" << r4 << " r3=" << r3; ResMgr::Stat(); // r4 C= from f(r3): trace debug to understand
} // m.~ResMgr is called after the destruction of local automatic objects to print the final statistics
// Dtor[R] Dtor[R] Dtor[R] Dtor[R]
// Resources Created = 8 Resources Released = 8 // printed from m.~ResMgr
```

- Note that ResMgr m is a global static object that is *created before and destroyed after* main()
- Track function call messages to understand the lifetimes of object

Move Semantics: MyResource: POD Resource Class
Copy & Move

```
class MyResource { // Representative resource class with copy-and-move support
    char *str = nullptr; // Resource pointer
public:
    MyResource(const char* s = nullptr) : str(ResMgr::Create(s)) // Param. & Defa. Ctor
    { cout << "Ctor[R] "; } // Creates resource
    MyResource(const MyResource& s) : str(ResMgr::Create(s.str)) // Copy Ctor
    { cout << "C-Ctor[R] "; } // Copy-Creates resource
    MyResource(MyResource&& s) noexcept : str(s.str) // Move Ctor
    { cout << "M-Ctor[R] "; s.str = nullptr; } // Moves resource
    MyResource& operator=(const MyResource& s) { cout << "C=[R] "; // Copy Assignment
        if (this != &s) { ResMgr::Release(str); str = ResMgr::Create(s.str); }
        return *this; // Releases and Copy-Creates resource
    }
    MyResource& operator=(MyResource&& s) noexcept { cout << "M=[R] ";  // Move Assignment
        if (this != &s) { ResMgr::Release(str); str = s.str; s.str = nullptr; }
        return *this; // Releases and Moves resource
    }
    ~MyResource() // Destructor
    { cout << "Dtor[R] "; ResMgr::Release(str); } // Releases resource
    friend ostream& operator<<(ostream& os, const MyResource& s) { // Streams resource value
        cout << ((s.str) ? s.str : "null"); return os; // Streams "null" for nullptr (no resource)
    }
};
MyResource f(MyResource s)      // Global function
{ cout << "f[R] "; return s; } // Uses call-by-value & return-by-value
```

Move Semantics: Application using *Copy & Move* MyResource

```
int main() { // Application to check resource behavior for copy-and-move support through
             // copy construction / assignment and move construction / assignment
    MyResource r1{ "ppd" }; // Ctor[R] r1=ppd Stat = (1, 0)
    cout << "r1=" << r1; ResMgr::Stat();

    MyResource r2{ r1 }; // C-Ctor[R] r2=ppd r1=ppd Stat = (2, 0)
    cout << "r2=" << r2 << " r1=" << r1; ResMgr::Stat();

    MyResource r3{ f(r2) }; // C-Ctor[R] f[R] M-Ctor[R] Dtor[R] r3=ppd r2=ppd Stat = (3, 0)
    cout << "r3=" << r3 << " r2=" << r2; ResMgr::Stat(); // r3 M-Ctor from f(r2): C-Ctor / Dtor for param

    r1 = r2; // C=[R] r1=ppd r2=ppd Stat = (4, 1)
    cout << "r1=" << r1 << " r2=" << r2; ResMgr::Stat();

    MyResource r4; // Ctor[R] r4=null Stat = (4, 1)
    cout << "r4=" << r4; ResMgr::Stat();

    r4 = f(r3); // C-Ctor[R] f[R] M-Ctor[R] M=[R] Dtor[R] Dtor[R] r4=ppd r3=ppd Stat = (5, 1)
    cout << "r4=" << r4 << " r3=" << r3; ResMgr::Stat(); // r4 M= from f(r3): Note M-Ctor in f(r3)
}
// Dtor[R] Dtor[R] Dtor[R] Dtor[R]
// Resources Created = 5 Resources Released = 5
```

- **Compared to copy-only, we created and released (8 – 5) = 3 resources less with copy-and-move**

Move Semantics: MyClass: UDT Resource Class - *Copy & Move* (**broken!**)


```
class MyClass { MyResource mRrc; // Resource object
public:
    MyClass() : mRrc("") // Defa. Ctor
    { cout << "D-Ctor[C] "; }
    MyClass(const MyResource& r) : mRrc(r) // Param. Ctor
    { cout << "Ctor[C] "; }
    MyClass(const MyClass& c) : mRrc(c.mRrc) // Copy Ctor
    { cout << "C-Ctor[C] "; }
    MyClass(MyClass&& c) noexcept : mRrc(c.mRrc) // Move Ctor
    { cout << "M-Ctor[C] "; }
    MyClass& operator=(const MyClass& c) { cout << "C=[C] "; // Copy Assignment
        if (this != &c) { mRrc = c.mRrc; }
        return *this;
    }
    MyClass& operator=(MyClass&& c) noexcept { cout << "M=[C] "; // Move Assignment
        if (this != &c) { mRrc = c.mRrc; }
        return *this;
    }
    ~MyClass() /* Destructor */ { cout << "Dtor[C] "; }
    friend ostream& operator<<(ostream& os, const MyClass& c) // Streams resource value
    { cout << c.mRrc; return os; }
};
MyClass f(MyClass s)            // Global function
{ cout << "f[C] "; return s; } // Uses call-by-value & return-by-value
```

Move Semantics: Application using *Copy & Move* MyClass

```
int main() {
    MyResource r1{ "ppd" }; // Ctor[R] r1=ppd Stat = (1, 0)
    cout << "r1=" << r1; ResMgr::Stat();

    MyClass c1{ r1 }; // C-Ctor[R] Ctor[C] c1=ppd Stat = (2, 0)
    cout << "c1=" << c1; ResMgr::Stat();

    MyClass c2{ f(c1) }; // C-Ctor[R] C-Ctor[C] f[C] C-Ctor[R] M-Ctor[C] Dtor[C] Dtor[R]
                         // c2=ppd c1=ppd Stat = (4, 1)
                         // c2 C-Ctor[C] from f(c1). Calls M-Ctor[C] yet copies resource
    cout << "c2=" << c2 << " c1=" << c1; ResMgr::Stat();

    c1 = f(c2); // C-Ctor[R] C-Ctor[C] f[C] C-Ctor[R] M-Ctor[C] M=[C] C=[R]
                // Dtor[C] Dtor[R] Dtor[C] Dtor[R]
                // c1=ppd c2=ppd Stat = (7, 4)
                // c1 C=[C] from f(c2). Calls M=[C] yet copies resource
    cout << "c1=" << c1 << " c2=" << c2; ResMgr::Stat();
}
// Dtor[C] Dtor[R] Dtor[C] Dtor[R] Dtor[R]
// Resources Created = 7 Resources Released = 7
```

Move Semantics: MyClass: : UDT Resource Class - *Copy & Move* (**fixed!**)


```
class MyClass { MyResource mRrc; // Resource object
public:
    MyClass() : mRrc("") // Defa. Ctor
    { cout << "D-Ctor[C] "; }
    MyClass(const MyResource& r) : mRrc(r) // Param. Ctor
    { cout << "Ctor[C] "; }
    MyClass(const MyClass& c) : mRrc(c.mRrc) // Copy Ctor
    { cout << "C-Ctor[C] "; }
    MyClass(MyClass&& c) noexcept : mRrc(std::move(c.mRrc)) // Move Ctor
    { cout << "M-Ctor[C] "; }
    MyClass& operator=(const MyClass& c) { cout << "C=[C] "; // Copy Assignment
        if (this != &c) { mRrc = c.mRrc; }
        return *this;
    }
    MyClass& operator=(MyClass&& c) noexcept { cout << "M=[C] "; // Move Assignment
        if (this != &c) { mRrc = std::move(c.mRrc); }
        return *this;
    }
    ~MyClass() /* Destructor */ { cout << "Dtor[C] "; }
    friend ostream& operator<<(ostream& os, const MyClass& c) // Streams resource value
    { cout << c.mRrc; return os; }
};
MyClass f(MyClass s)           // Global function
{ cout << "f[C] "; return s; } // Uses call-by-value & return-by-value
```

Move Semantics: Application using *Copy & Move* MyClass

```
int main() {
    MyResource r1{ "ppd" }; // Ctor[R] r1=ppd Stat = (1, 0)
    cout << "r1=" << r1; ResMgr::Stat();

    MyClass c1{ r1 }; // C-Ctor[R] Ctor[C] c1=ppd Stat = (2, 0)
    cout << "c1=" << c1; ResMgr::Stat();

    MyClass c2{ f(c1) }; // C-Ctor[R] C-Ctor[C] f[C] M-Ctor[R] M-Ctor[C] Dtor[C] Dtor[R]
                         // c2=ppd c1=ppd Stat = (3, 0)
                         // c2 C-Ctor[C] from f(c1). Calls M-Ctor[C] and does not copy. FIXED
    cout << "c2=" << c2 << " c1=" << c1; ResMgr::Stat();

    c1 = f(c2); // C-Ctor[R] C-Ctor[C] f[C] M-Ctor[R] M-Ctor[C] M=[C] M=[R]
                // Dtor[C] Dtor[R] Dtor[C] Dtor[R]
                // c1=ppd c2=ppd Stat = (4, 1)
                // c1 C=[C] from f(c2). Calls M-Ctor[C] and does not copy. FIXED
    cout << "c1=" << c1 << " c2=" << c2; ResMgr::Stat();
}
// Dtor[C] Dtor[R] Dtor[C] Dtor[R] Dtor[R]
// Resources Created = 4 Resources Released = 4
```

- **Compared to copy-and-move without `std::move`, we created and released $(7-4)=3$ resources less with `std::move`**

Sources:

- Universal References in C++11 – Scott Meyers, isocpp.org, 2012
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++
- Understanding Move Semantics and Perfect Forwarding: Part 1, Part 2: Rvalue References and Move Semantics, Part 3: Perfect Forwarding, Drew Campbell, 2018

Type Deduction in C++11/C++14: Universal References

Universal References

- T&& really is a magical reference type!
  - For lvalue arguments, T&& becomes T& => lvalues can bind
  - For rvalue arguments, T&& remains T&& => rvalues can bind
  - For const/volatile arguments, const/volatile becomes part of T
  - T&& parameters can bind *anything*
- Two conceptual meanings for T&& syntax:
  - **Rvalue reference**. Binds rvalues only

```
void f(Widget&& param);      // takes only non-const rvalue
```

  - **Universal reference**. Binds lvalues and rvalues

```
template<typename T>
void f(T&& param);          // takes lvalue or rvalue, const or non-const
```

    - Really an rvalues reference in a reference-collapsing context

auto&& $\equiv$ T&&

- auto type deduction $\equiv$ template type deduction, so auto&& variables are also universal references:

```
int calcVal();
int x;

auto&& v1 = calcVal(); // deduce type from rvalue => v1's type is int&&

auto&& v2 = x;         // deduce type from lvalue => v2's type is int&
```

- Note that decltype()&& does not behave like a universal references as it does not use template type deduction:

```
decltype(calcVal()) v3; // deduced type is int

decltype(x) v4;         // deduced type is int
```

Rvalue References vs. Universal References

- Read code carefully to distinguish them
  - Both use && syntax: Occus after a POD or UDT for Rvalue References, but after type variable T for Universal References
  - Type deduction for T for Universal References
  - Behavior is different:
    - Rvalue references bind only rvalues
    - Universal references bind lvalues *and* rvalues
      - that is, may become either T& or T&&, depending on initializer
- Consider std::vector:

```
template<class T, class Allocator=allocator<T>> // from C++11 Standard
class vector { public: ...
    void push_back(const T& x);          // lvalue reference
    void push_back(T&& x);               // rvalue reference!
    template<class... Args>
    void emplace_back(Args&&... args);   // universal reference
    ...
};
```

Type Deduction in C++11/C++14: Perfect Forwarding

Sources:

- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++
- Perfect Forwarding, modernescpp.com, 2016
- Understanding Move Semantics and Perfect Forwarding: Part 1, Part 2: Rvalue References and Move Semantics, Part 3: Perfect Forwarding, Drew Campbell, 2018

Type Deduction in C++11/C++14: Perfect Forwarding

Perfect Forwarding

- Goal: one function that *does the right thing:*
  - *Copies* lvalue args
  - *Moves* rvalue args
- Solution is a perfect forwarding function:
  - Templatized function forwarding `T&&` params to members
- **What is Perfect Forwarding?**
  - Perfect forwarding allows a template function that accepts a set of arguments to forward these arguments to another function whilst retaining the lvalue or rvalue nature of the original function arguments
- Let us check an example

Perfect Forwarding Example: (broken)

```
#include <iostream>
class Data { int i; public: Data(): i(0) { } }; // a UDT

void g(const int&) { std::cout << "int& in g" << "; "; }  // binds with lvalue parameter
void g(int&&)       { std::cout << "int&& in g" << "; "; } // binds with rvalue parameter

void h(const Data&) { std::cout << "Data& in h" << std::endl; }  // binds with lvalue parameter
void h(Data&&)       { std::cout << "Data&& in h" << std::endl; } // binds with rvalue parameter

template<typename T1, typename T2>
void f(T1&& p1, T2&& p2) { // universal ref. gets lvalue or rvalue from arg by template type deduction
    g(p1); // always binds with lvalue parameter as p1 is an lvalue in f
    h(p2); // always binds with lvalue parameter as p2 is an lvalue in f
}
int main() { int i { 0 }; Data d;
    f(i, d);                       // (lvalue, lvalue) binds with int& in g; Data& in h
    f(std::move(i), d);            // (rvalue, lvalue) binds with int& in g; Data& in h
    f(i, std::move(d));            // (lvalue, rvalue) binds with int& in g; Data& in h
    f(std::move(i), std::move(d)); // (rvalue, rvalue) binds with int& in g; Data& in h
}
```

- Lvalue arg passed to `p1` ⇒ `g(const int&)` receives Lvalue
- Rvalue arg passed to `p1` ⇒ `g(int&&)` receives Lvalue
- Lvalue arg passed to `p2` ⇒ `h(const Data&)` receives Lvalue
- Rvalue arg passed to `p2` ⇒ `h(Data&&)` receives Lvalue

Perfect Forwarding Example: (fixed) by std::forward

```
#include <iostream>
class Data { int i; public: Data(): i(0) { } }; // a UDT

void g(const int&) { std::cout << "int& in g" << "; "; }  // binds with lvalue parameter
void g(int&&)      { std::cout << "int&& in g" << "; "; } // binds with rvalue parameter

void h(const Data&) { std::cout << "Data& in h" << std::endl; }  // binds with lvalue parameter
void h(Data&&)      { std::cout << "Data&& in h" << std::endl; } // binds with rvalue parameter

template<typename T1, typename T2>
void f(T1&& p1, T2&& p2) { // universal ref. gets lvalue or rvalue from arg by template type deduction
    g(std::forward<T1>(p1)); // std::forward forwards lvalue arg to lvalue param and
    h(std::forward<T2>(p2)); // rvalue arg to rvalue param
}
int main() { int i { 0 }; Data d;
    f(i, d);                          // (lvalue, lvalue) binds with int& in g; Data& in h
    f(std::move(i), d);               // (rvalue, lvalue) binds with int&& in g; Data& in h
    f(i, std::move(d));               // (lvalue, rvalue) binds with int& in g; Data&& in h
    f(std::move(i), std::move(d)); // (rvalue, rvalue) binds with int&& in g; Data&& in h
}
```

- Lvalue arg passed to p1 ⇒ `g(const int&)` receives Lvalue
- Rvalue arg passed to p1 ⇒ `g(int&&)` receives Rvalue
- Lvalue arg passed to p2 ⇒ `h(const Data&)` receives Lvalue
- Rvalue arg passed to p2 ⇒ `h(Data&&)` receives Rvalue

Perfect Forwarding

- Despite `T&&` parameters, code fully type-safe:
- Type compatibility verified upon instantiation
  - Only `int`-compatible types valid for call to `g()`
  - Only `Data`-compatible types valid for call to `h()`. For example in the context of

```
...
class DerivedData: public Data { public: DerivedData(): Data() { } };
...
int main() { ... DerivedData d; ... }
```

The code works exactly as before. Whereas for

```
...
class OtherData { int i; public: OtherData(): i(0) { } }; // another UDT
...
int main() { ... OtherData d; ... }
```

The code fails compilation: `error: no matching function for call to h(OtherData&)`

Perfect Forwarding

- The flexibility can be removed via `static_assert` as follows:

```
...
template<typename T1, typename T2>
void f(T1&& p1, T2&& p2) {
    // Asserts that T2 must be of type Data
    static_assert(std::is_same< typename std::decay<T2>::type, Data >::value,
        "T2 must be Data");

    g(std::forward<T1>(p1)); // T1 too may be asserted, if needed
    h(std::forward<T2>(p2));
}
...
class DerivedData: public Data { public: DerivedData(): Data() { } };
...
int main() { ... DerivedData d; ... }
```

Compile-time error: `error: static assertion failed: T2 must be Data`

Type Deduction in C++11/C++14: Perfect Forwarding: Examples

Perfect Forwarding Example 1: Modified from slide M07.160


```
#include <iostream>
class Data { int i; public: Data(int i = 5): i(i) { }
    operator int() { return i; }                    // cast to int
    Data& operator++() { ++i; return *this; } // pre-increment operator
    Data& operator--() { --i; return *this; } // pre-decrement operator
};
void g(int& a) { std::cout << "int& in g: " << ++a << "; ";  }          // binds non-const lvalue param
void g(int&& a) { std::cout << "int&& in g: " << --a << "; "; }         // binds rvalue param

void h(Data& a) { std::cout << "Data& in h: " << ++a << std::endl; }   // binds non-const lvalue param
void h(Data&& a) { std::cout << "Data&& in h: " << --a << std::endl; } // binds rvalue param

template<typename T1, typename T2>
void f(T1&& p1, T2&& p2) {
    g(...); // called on p1 with or without std::forward
    h(...); // called on p1 with or without std::forward
}
int main() { int i { 0 }; Data d;
    f(i, d);          // (lvalue, lvalue)
    f(5, d);          // (rvalue, lvalue)
    f(i, Data(7)); // (lvalue, rvalue)
    f(5, Data(7)); // (rvalue, rvalue)
}
```

Without std::forward

```
int& in g: 1; Data& in h: 6
int& in g: 6; Data& in h: 7
int& in g: 2; Data& in h: 8
int& in g: 6; Data& in h: 8
```

With std::forward

```
int& in g: 1; Data& in h: 6
int&& in g: 4; Data& in h: 7
int& in g: 2; Data&& in h: 6
int&& in g: 4; Data&& in h: 6
```

Perfect Forwarding Example 2: Generic Factory Method/1

- Let us write a *generic factory method* that should be able to create each arbitrary object. That means that the function should have the following characteristics:
  - Can take an arbitrary number of arguments
  - Can accept lvalues and rvalues as an argument
  - Forwards it arguments identical to the underlying constructor

```
#include <iostream>

template <typename T, typename Arg>  // For efficiency reasons, the function template should
T CreateObject(Arg& a) {             // take its arguments by a non-const lvalue reference
    return T(a);
}
int main() {
    int five=5; // lvalues
    int myFive= CreateObject<int>(five);
    std::cout << "myFive: "  << myFive << std::endl;

    int myFive2= CreateObject<int>(5); // rvalues: error: cannot bind non-const lvalue reference
                                       // of type int& to an rvalue of type int
    std::cout << "myFive2: " << myFive2 << std::endl;
}
```

Perfect Forwarding Example 2: Generic Factory Method/2

- To solve the compilation issues, we can go one of two ways:
  - Change the non-const lvalue reference to a const lvalue reference (that can bind an rvalue). But that is not perfect, because we cannot change the function argument, if needed
  - Overload the function template for a const lvalue reference and a non-const lvalue reference. That is preferred

```
#include <iostream>

template <typename T, typename Arg> T CreateObject(Arg& a) { return T(a); }       // binds lvalues

template <typename T, typename Arg> T CreateObject(const Arg& a) { return T(a); } // binds rvalues

int main() {
    int five = 5; // lvalues
    int myFive = CreateObject<int>(five);
    std::cout << "myFive: "  << myFive << std::endl;  // myFive: 5

    int myFive2 = CreateObject<int>(5); // rvalues
    std::cout << "myFive2: " << myFive2 << std::endl; // myFive2: 5
}
```

Perfect Forwarding Example 2: Generic Factory Method/3

- The solution has two conceptual issues:
  - To support n arguments, we need to overload $2^n + 1$ variations of `CreateObject<T>(...)`. "+1" for the function `CreateObject<T>()` without any argument
  - Without the overload, the forwarding problem would appear for rvalue arguments as they will be *copied* instead of being *moved*
- So we need to use universal reference in `CreateObject<T>(...)` with `std::forward`

```
#include <iostream>

template <typename T, typename Arg>
T CreateObject(Arg&& a) {            // Universal reference
    return T(std::forward<Arg>(a)); // std::forward
}
int main() {
    int five = 5; // lvalues
    int myFive = CreateObject<int>(five);
    std::cout << "myFive: "  << myFive << std::endl;  // myFive: 5

    int myFive2 = CreateObject<int>(5); // rvalues
    std::cout << "myFive2: " << myFive2 << std::endl; // myFive2: 5
}
```

Perfect Forwarding Example 2: Generic Factory Method/4

- `CreateObject<T>()` needs exactly one argument perfectly forwarded to the constructor
- For arbitrary number of arguments, we need a *variadic template* (TBD later)

```
#include <iostream>
#include <string>
#include <utility>
template <typename T, typename ... Args> // Variadic Templates can get an arbitrary number of arguments
    T CreateObject(Args&& ... args) { return T(std::forward<Args>(args)...); }
int main() {
    int five = 5, myFive = CreateObject<int>(five); // lvalues
    std::cout << "myFive: "  << myFive << std::endl;                      // myFive: 5
    std::string str { "Lvalue" }, str2 = CreateObject<std::string>(str);
    std::cout << "str2: " << str2 << std::endl;                           // str2: Lvalue

    int myFive2 = CreateObject<int>(5); // rvalues
    std::cout << "myFive2: " << myFive2 << std::endl;                     // myFive2: 5
    std::string str3 = CreateObject<std::string>(std::string("Rvalue"));
    std::cout << "str3: " << str3 << std::endl;                           // str3: Rvalue
    std::string str4 = CreateObject<std::string>(std::move(str3));
    std::cout << "str4: " << str4 << std::endl;                           // str4: Rvalue

    double doub = CreateObject<double>(); // Arbitrary number of args
    std::cout << "doub: " << doub << std::endl;                           // doub: 0
    struct Data { Data(int i, double d, std::string s) { } } d = CreateObject<Data>(2011, 3.14, str4);
}
```

Perfect Forwarding Example 3: apply Functor/1

- Let us design an apply functor that would take a function and its arguments and apply the function on the arguments

```
template<typename F, typename... Ts> // Using variadic template (TBD later)
auto apply(std::ostream& os, F&& func, Ts&&... args)
    -> decltype(func(args...)) { // may not preserves rvalue-ness
    os << "Forwarding:: ";
    return func(args...);          // may not preserves rvalue-ness
}
```

- args... are lvalues, but apply's caller may have passed rvalues:
    - Templates can distinguish rvalues from lvalues
    - apply might call the wrong overload of func

```
class Data { };
Data myData() { return Data(); }

class DataDispatcher { public:
    void operator()(const Data&) { std::cout << "operator()(const Data&) called\n\n"; } // takes lvalue
    void operator()(Data&&) { std::cout << "operator()(Data&&) called\n\n"; }           // takes rvalue
};
int main() { Data d = myData();
    apply(std::cout, DataDispatcher(), d);        // Forwarding:: operator()(const Data&) called
    apply(std::cout, DataDispatcher(), myData()); // Forwarding:: operator()(const Data&) called
                                                  // rvalue forwarded as lvalue!
}
```

Perfect Forwarding Example 3: apply Functor/2

- Naturally, perfect forwarding solves

```
template<typename F, typename... Ts>
auto apply(std::ostream& os, F&& func, Ts&&... args) // return type is same as func's on original args
    -> decltype(func(std::forward<Ts>(args)...)) { // preserves lvalue-ness / rvalue-ness
    os << "Forwarding:: ";
    return func(std::forward<Ts>(args)...);        // preserves lvalue-ness / rvalue-ness
}
...
int main() { Data d = myData();
    apply(std::cout, DataDispatcher(), d);         // Forwarding:: operator()(const Data&) called

    apply(std::cout, DataDispatcher(), myData()); // Forwarding:: operator()(Data&&) called
}
```

- With return type deduction [C++14]

```
template<typename F, typename... Ts>
decltype(auto) apply(std::ostream& os, F&& func, Ts&&... args) { // return type deduction
    os << "Forwarding:: ";
    return func(std::forward<Ts>(args)...);
}
```

Perfect Forwarding Example 3: apply Functor/3

- Perfect forwarding works perfectly with mixed bindings as well

```
#include <iostream>
using namespace std;
class Data { };
Data myData() { return Data(); }

class DataDispatcher { public: // mixed binding for two parameters
    void operator()(const Data&, const Data&){ cout<< "operator()(const Data&, const Data&) called\n\n"; }
    void operator()(const Data&, Data&&){ cout<< "operator()(const Data&, Data&&) called\n\n"; }
    void operator()(Data&&, const Data&){ cout<< "operator()(Data&&, const Data&) called\n\n"; }
    void operator()(Data&&, Data&&){ cout<< "operator()(Data&&, Data&&) called\n\n"; }
};
template<typename F, typename... Ts>
auto apply(ostream& os, F&& func, Ts&&... args) -> decltype(func(forward<Ts>(args)...)) {
    return func(forward<Ts>(args)...);
}
int main() {
    Data d = myData();
    apply(cout, DataDispatcher(), d, d);                 // operator()(const Data&, const Data&) called
    apply(cout, DataDispatcher(), d, myData());          // operator()(const Data&, Data&&) called
    apply(cout, DataDispatcher(), myData(), d);          // operator()(Data&&, const Data&) called
    apply(cout, DataDispatcher(), myData(), myData());  // operator()(Data&&, Data&&) called
}
```

Type Deduction in C++11/C++14: `std::forward`

Sources:

- Universal References in C++11 – Scott Meyers, isocpp.org, 2012
- std::forward, cppreference.com
- Quick Q: What's the difference between std::move and std::forward?, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++

Type Deduction in C++11/C++14: `std::forward`

std::forward

- Let us relook at:

```
template<typename T1, typename T2>
void f(T1&& p1, T2&& p2) { ... h(std::forward<T2>(p2)); }
```

  - T a reference (that is,T is T&) ⇒ lvalue was passed to p2
    - ▷ std::forward<T>(p2) should return lvalue
  - T a non-reference (that is, T is T) ⇒ rvalue was passed to p2
    - ▷ std::forward<T>(p2) should return rvalue
- std::forward is provided in <utility> for this
  - Applicable only to *function templates*
  - *Preserves arguments' lvalue-ness / rvalue-ness / const-ness* when forwarding them to other functions
- Let us take a look at the implementation

std::forward

- C++11 implementations:

```
template<typename T>        // For lvalues (T is T&);
T&&                         // return lvalue reference
forward(typename remove_reference<T>::type& t) noexcept
{ return static_cast<T&&>(t); }

template<typename T>        // For rvalues (T is T);
T&&                         // return rvalue reference
forward(typename remove_reference<T>::type&& t) noexcept
{ return static_cast<T&&>(t); }
```

- By design, param type disables type deduction ⇒ callers must specify T:

```
template<typename T1, typename T2> void f(T1&& p1, T2&& p2)
{ g(std::forward(p1)); ... } // error! Cannot deduce T1 in call to std::forward

template<typename T1, typename T2> void f(T1&& p1, T2&& p2)
{ g(std::forward<T1>(p1)); ... } // fine
```

Type Deduction in C++11/C++14: Move is an Optimization of Copy

Sources:

- Scott Meyers on C++
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

Move is an Optimization of Copy

Copy Only

- Move requests for copyable types w/o move support yield copies:

```
class MyResource { public: // w/o move support
    MyResource(const MyResource&); // copy ctor

};
class MyClass { public: // with move support
    MyClass(MyClass&& src)  // move ctor
    // request to move r's value
    : w(std::move(src.r)) { ... }
    private: MyResource r; // no move support
};
```

src.r is copied to r:

- std::move(src.r) returns an rvalue of type MyResource
- That rvalue is passed to MyResource's copy constructor

Copy & Move

- If MyResource adds move support:

```
class MyResource { public: // with move support
    MyResource(const MyResource&);      // copy ctor
    MyResource(MyResource&&) noexcept; // move ctor
};
class MyClass { public: // with move support
    MyClass(MyClass&& src) noexcept
    // request to move r's value
    : w(std::move(src.r)) { ... }
    private: MyResource r;
};
```

src.r is moved to r:

- std::move(src.r) returns an rvalue of type MyResource
- That rvalue is passed to MyResource's move constructor via normal overloading resolution

Move is an Optimization of Copy

- Implications:
  - *Giving classes move support can improve performance even for* **move-unaware code**
    - Copy requests for rvalues may silently become moves
  - *Move requests safe for types without explicit move support*
    - Such types perform copies instead
      - For example, all built-in types (POD)
  - *Move support may exist even if copy operations do not*
    - For example, **Move-only types** like `std::thread` and `std::unique_ptr` that are *moveable*, but not *copyable*
  - *Types should support move when moving cheaper than copying*
    - Libraries use moves whenever possible (for example, STL)
- In short:
  - **Give classes move support when moving faster than copying**
  - **Use `std::move` for lvalues that may safely be moved from**

Move is an Optimization of Copy: Use Beyond Construction / Assignment

- Move support useful for other functions, e.g., setters:

```
class MyList { public:
    ...
    void setID(const std::string& newId)          // copy param
    { id = newId; }
    void setID(std::string&& newId) noexcept      // move param
    { id = std::move(newId); }
    void setVals(const std::vector<int>& newVals) // copy param
    { vals = newVals; }
    void setVals(std::vector<int>&& newVals)      // move param
    { vals = std::move(newVals); }
    ...
private:
    std::string id;
    std::vector<int> vals;
};
```

- **Note**:
  - As the move `operator=` of `std::string` is `noexcept`, `setId` is declared `noexcept`
  - Whereas `setVals` is not declared `noexcept`, as the move `operator=` of `std::vector` is not declared `noexcept`

Compiler Generated Move Operations

- Move constructor and move `operator=` are *special*:
  - Generated by compilers under appropriate conditions
- Conditions:
  - *All data members and base classes are movable*
    - ▷ Implicit move operations move everything
    - ▷ Most types qualify:
      - All built-in types (move $\equiv$ copy).
      - Most standard library types (for example, all containers).
  - Generated operations likely to maintain class invariants
    - ▷ *No user-declared copy or move operations*
      - Custom semantics for any $\Rightarrow$ default semantics inappropriate
      - Move is an optimization of copy
    - ▷ *No user-declared destructor*
      - Often indicates presence of implicit class invariant
- More on this later in the Module discussing `default` and `delete`

Compiler Generated Move Operations: Custom Deletion ⇒ Custom Copying

```
class Widget { private:
    std::vector<int> v;
    std::set<double> s;
    std::size_t sizeSum;
public:
    ~Widget() { assert(sizeSum == v.size()+s.size()); }
  ...
};
```

- If `Widget` had implicitly-generated move operations:

```
std::vector<Widget> vw;
Widget w;
...                              // put stuff in w's containers
vw.push_back(std::move(w));      // move w into vw
...                              // no use of w
```

- *User-declared dtor ⇒ no compiler-generated move ops for* `Widget`

Compiler Generated Move Operations: Custom Moving ⇒ Custom Copying

copyable & movable type

```
class Widget1 { private:
    std::u16string name; // copyable/movable type
    long long value;     // copyable/movable type
public: explicit Widget1(std::u16string n);


}; // implicit copy/move ctor
   // implicit copy/move operator=
```

copyable type; not movable

```
class Widget2 { private:
    std::u16string name;
    long long value;
public: explicit Widget2(std::u16string n);
    // user-declared copy ctor
    Widget2(const Widget2& rhs);
}; // => no implicit move ops
   // implicit copy operator=
```

- *Declaring a move operation prevents generation of copy operations*
  - Custom move semantics ⇒ custom copy semantics
    - ▷ **Move is an optimization of copy**

```
class Widget3 { private:                // movable type; not copyable
    std::u16string name;
    long long value;
public:
    explicit Widget3(std::u16string n);
    Widget3(Widget3&& rhs) noexcept;    // user-declared move ctor => no implicit copy ops
    Widget3&                            // user-declared move op=
    operator=(Widget3&& rhs) noexcept; // => no implicit copy ops
};
```