Note 4

Variables

Yunheung Paek
Associate Professor
Software Optimizations and Restructuring Lab.
Seoul National University

Topics

- Name
- Assignment
- l-value and r-value
- Scope
- Binding

Variables and assignments

- Components of a variable
  1. name : identifier composed of letters
  2. memory location : bound to the variable
  3. current value : stored in the location
  4. data type : static/dynamic binding
  5. scope : static/dynamic
  6. life time : interval during which a location is bound to the variable
- An assignment changes the contents of some components of a variable.

Scheme

```
> (set! x 10)
10
> (set! x 4.5)
4.5
```

//the type is an integer of value 10

//now, the type is a real number of value 4.5

location (l-value) and value (r-value)

- The ***location*** of the variable is regarded as a kind of value owned by the variable.

  → so, we call the location the I-value of the variable.

- To make a distinction, the ***real value*** of the variable is called the r-value.

  Ex: `char z = 'o';` // the l-value of z is the address of z and the r-value is 'o'

- A rule of assignments:

  "The left-hand side of an assignment should have the l-value, and its right-hand side should have the r-value."

C++

```
float x;          // create real variable x with the l-value in which the r-value is undefined
float y = 4.1;    // create real variable y with the r-value defined by storing 4.1 to the l-value of y
x = y;            // store the r-value of y into the l-value of x
y = y * 3.0;      // store the r-value of y times 3.0 into the l-value of y
```

- When different variables have the same l-value, it is called *aliasing*. ➔ Ex:

```
char d;
char & c = d;   // c and d share the same location
```

Properties of l-values and r-values

- **referencing**: the operation of getting the l-value of a variable

  Ex: `char* p = &c;` → *note the graphical interpretation!*

- **dereferencing**: the operation of going from a reference to the r-value it refers

  → In C++, the right-hand side is always dereferenced **once**.

```
char c, d;
char *p, **q, **r;
c = d;          // OK! dereferenced once
q = r;          // OK! dereferenced once
d = p;          // error! Should be dereferenced twice
p = c;          // error! no dereferencing necessary
r = c;          // error! no dereferencing sufficient
```

Properties of l-values and r-values

- Some expressions, such as *id*, *array reference* and *dereference*, have both l-values and r-values.
  - All other expressions have r-values only.
  - Thus, these other expressions can **NOT** appear on the left-hand side of assignments.

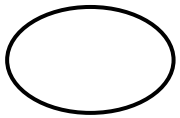
```
4.5 = 10.1;                    // illegal - integers have no l-values
“jane” = y;                    // illegal - strings have no l-values
y + 1 = a[i];                  // illegal - arithmetic expressions have no l-values
x = 10.1;                      // legal - id expressions have l-values
a[i] = y + 1                   // legal - array references have l-values
employee.name = “jane”         // legal
*p = foo(x,y)                  // legal - dereference expressions have l-values
foo(x,y) = 1.1;                // maybe legal, maybe illegal
```

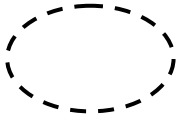
gnu c++ says error: ‘non-lvalue in assignment’

Graphical notation for variables


name

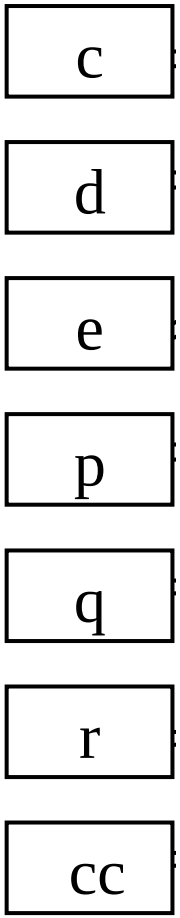

Location that can contain an ordinary data value


Location that can contain a reference (address) value


Location that can contain a reference-reference value

Graphical notation for variables

```
char c = 'w';
char d = c;
char& e = d;
char* p = &c;
char** q = &p;
char** r = q;
const char cc = 'w'
```

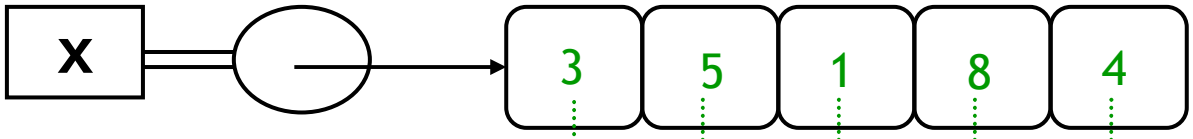


Array variables

- In Pascal, when you declare

```
x, y : array [1..5] of integer;
```

the compiler creates not only the storage for 5 integers, but also a pointer to its beginning address.

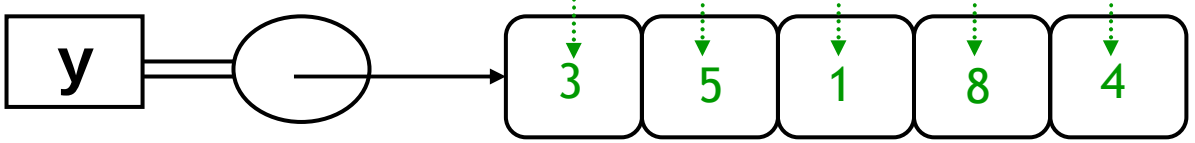


When we have an array assignment

```
y = x;
```

copy

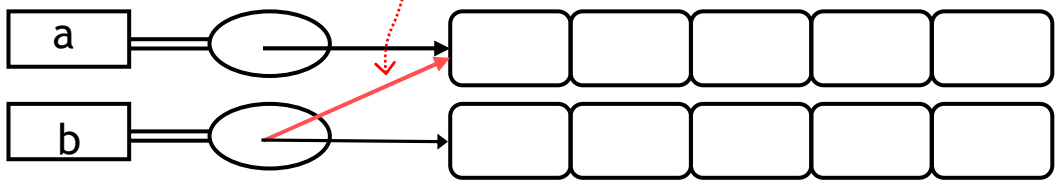
the whole contents of the array on right-hand side are copied to the location of the array on the left-hand side.



Array variables

- In C++, an array is really a pointer. So, arrays and pointers can be mixed. Thus, after b = a, we should have ...

```
int a[5], b[5];
```



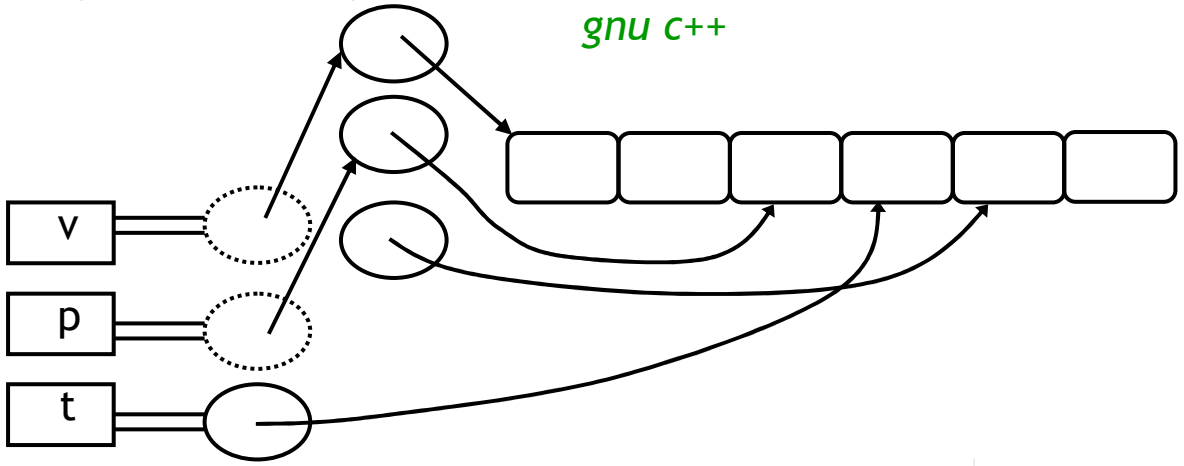
But in practice, this is totally implementation dependent
SUN C++: illegal
GNU C++: copy

like Pascal

- In C++, a multi-dimensional array is simulated with arrays of arrays.

Thus, given `int c[5], d[3];` `d = c` *would be illegal with gnu c++*

```
int v[3][2];
int **p = v + 1;
int *t = *p + 1;
```



Name conflicts

- In languages, a name can be used to mean several different objects. ➔ Ex) Ford: : a name of a man, a car, or a company …
- Using the same name to represent different objects in different places causes a **name conflict** and potentially ambiguity of the language.
- But, name conflicts are often necessary to improve naturalness of a language and readability/writability.

Ford drives a car.
They work for the Ford.
She drives a Ford.
→ *How do we know which Ford means which in each sentence?*

Ford1 drives a car.
They work for the Ford2.
She drives a Ford3.
→ *OK. no more naming conflict! So, the meaning is clearer. But do you like it?*

Scope

- Ambiguity due to naming conflicts can be resolved by associating names with the **environments** where each name is defined.
  → *Jane in Ohio, Jane in Maine, Jane in Virginia*
- In a programming language, a **scope** is a program environment in which names are defined or declared.
  → procedures and blocks (Pascal, C), lambda expressions (Scheme)
- Through a declaration within some scope, a name is bound to a variable with certain attributes, and the variable is called a **bound variable**.
- A variable which is not bound in the scope is called a **free variable**.

Scope

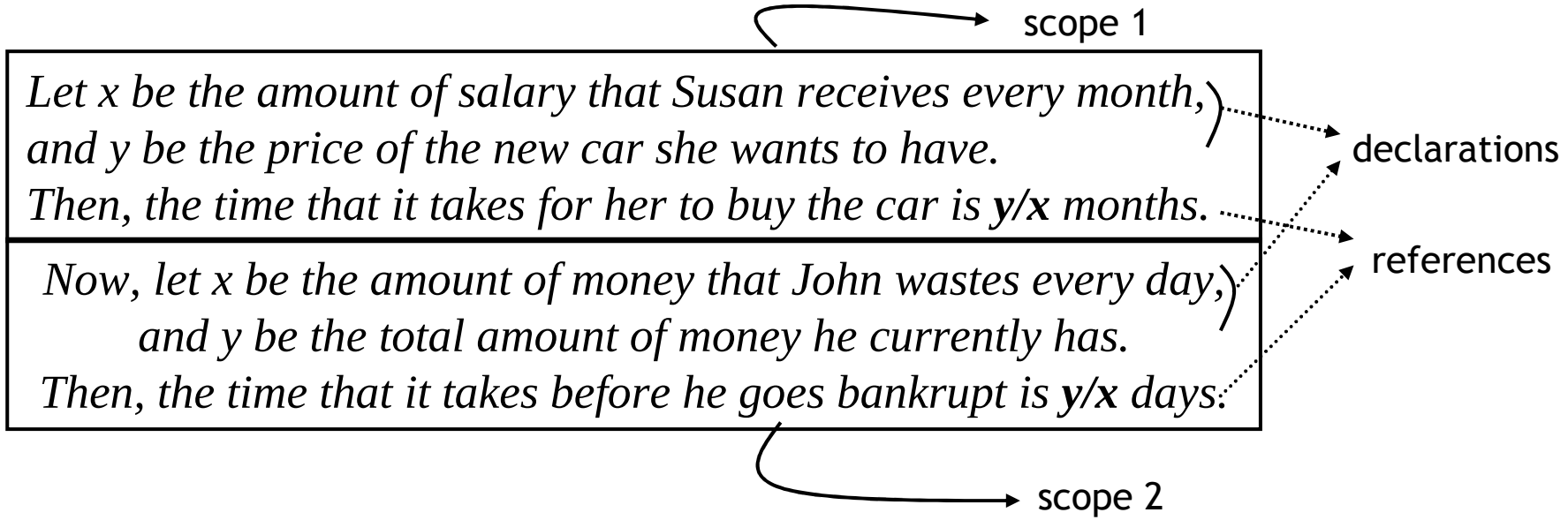
```
{                                  // beginning of a new environment/scope
   int a[10]; int i; char c;       // declarations of three names
   c …                             // a reference to 'c': bound to a character variable
   … a[i]                          // references to 'a' and 'i': bound to integer variables
   x …                             // a reference to the free variable 'x' within this scope
}                                  // end of a scope
… c …                              // The name no longer represents a character variable.
                                   // Then, what is it now?
```

→ *A binding of a name is visible and effective inside the scope where the name is declared.*

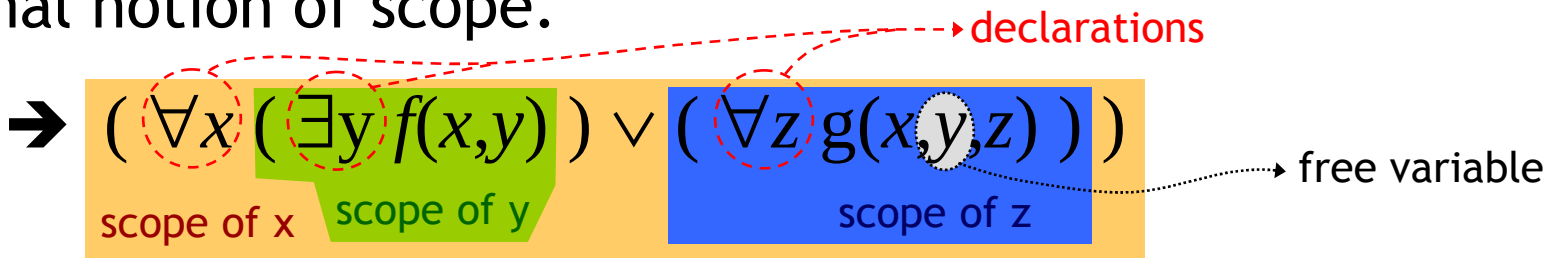
- The idea of a scope is to limit the boundary of a declaration of a name.
- Outside the boundary of a declaration, that declaration and binding should not be visible.

Example: scopes in mathematics

- The notion of a scope also appears in mathematics.



- Existential, ∃, and universal, ∀, quantifiers provide a formal notion of scope.



→ A declaration of x makes x a bound variable.