

Naming (II)

Variables

- A variable represents the **binding** of an **identifier** to a **memory address**.
 - The binding of a variable and address **requires four pieces of information**:
 - **Identifier** string
 - **format** of identifier strings is **defined by** the **syntax**
 - **Address** (implementation specific)
 - **uniquely identify** the actual memory location
 - **Type**
 - Even a language **without explicit variable types**, the compiler or interpreter must, **internally, handle typing**. Otherwise, it does not know how to implement the basic operators on the data stored at the address.
 - Because the **data is just bits**, and it is the **type of manipulation** applied to the bits that **determines the semantic meaning** (if any) of the bits.
 - **Value** (memory address or value)
-

l-value and r-value

- A **variable provides** both a **memory address** and a **value**, a language has to be clear about which to use in an expression.
 - If a variable's **identifier** is on the **left hand side** of an expression, it must provide a **memory address**.
 - If a variable's **identifier** is on the **right hand side** of an expression, it must provide a **value**.
 - **In some cases** the **value is**, in fact, a **memory address**, depending upon the language.
- The **convention** is that the **memory address meaning** of a variable is called **l-value**, and the **value meaning** of a variable is called **r-value**.
- Most languages do not require/allow explicit dereferencing of variables, although some language (C) provide a suite of tools for doing so.
- C Example

```
int x;
int y;
int *z;

// l-value and r-value of x
x = x + 1;

// l-value of y, r-value of x
y = x + 1;

// l-value of z, get l-value (address) of x and convert it to an r-value
z = &x;

// l-value of x, dereferenced z provides a r-value, change value of x
x = *z + 1;
```

```
// dereferenced z provides a l-value, r-value of y
*z = y;

// l-value of z, get l-value of y (address) and convert it to an r-value, change value of z
z = &y + 4;

// l-value of x, r-value of z and it's a memory address
x = (int)z
```

Scope vs Lifetime

- The **scope** of an identifier is the **part of program** in which the **identifier can be used to access its binding entity**.
 - C, Java, Python use **static scoping** (lexical scoping). A name is bound to a collection of statements in terms of its **position in the source program**.
 - Good readability and better compile-time checking
 - Perl uses **dynamic scoping**. A name is bound to its most recent declaration in terms of the **program's execution history**.
 - Generally not used, since it makes type checking difficult and is prone to errors
 - **Local scope**: accessible within function or statement block from point of declaration until the end of the block
 - **Class scope**: accessible by class members
 - **File scope**: accessible within the current file (a global variable in a C file has file scope)
 - **Global scope**: accessible everywhere unless "hidden"
- REF: <https://www.csee.umbc.edu/~chang/cs202.f15/Lectures/modules/m05-scope/slides.php?print>
- **Lifetime** of an identifier is the **time interval** during which the **identifier has been allocated a block of memory**.
 - Once the lifetime of a variable ends, the value the variable held is lost. Even if the identifier is recreated, it may be bound to a different memory location.
- Today, most PLs **link lifetime and scope**. Variables do not generally retain their value once they go out of scope because their identifiers lose the binding to a specific memory address.
- Ways to modify the lifetime/scope of variables
 - **static**: a static variable's lifetime is the **entire duration of the program execution**.
 - **extern**: **propagate global variables to other compilable units and make the global variable has global scope**. Global variables can be hidden from other compilable units by making them as static variables. (Used in C header file)
- Some C code below to demonstrate the scope

Q1: will the value of i and si change in each iteration? [si change, i not]

Q2: can the program change ci's value? [No]

The lifetime of static variable is the entire duration of the program execution.

The lifetime of const is function block. A new const int is declared every time the foo function is invoked.

```
/**
 * staticInC.c
 * Ying Li
 */

#include <stdio.h>

void foo () {
    int i = 10;
    static int si = 10;
    const int ci = 10;

    i += 10;
    si += 10;

    printf("i = %3d, si = %3d, ci = %3d \n", i, si, ci);
}

int main () {
    for (int i = 0; i < 10; i++)
        foo();

    return 0;
}
```