CS 315 – Programming Languages
Names, Scopes, Bindings

Names

Names of identifiers, functions, variables, types, classes, etc.

- Old languages put limits on the name length

  E.g.: C99 uses up to 63 chars, Fortran 95 uses up to 31 chars.

- Case sensitivity of names is also language specific.

  Circle circle = new Circle();
  // the above line relies on case-sensitivity

- Some languages make use of *sigils*: Symbols attached to variable names, typically showing the type

  E.g.: Perl: $x is a scalar variable, @x is a list variable

- Special words
  - **Keywords**: special in only certain contexts

    E.g.: Fortran

Integer Apple	Integer is a typename in a variable declaration
Integer = 4	Integer is a variable name in an assignment context

    This may cause confusion:

Integer Real	Integer variable with name Real
Real Integer	Real variable with name Integer

  - **Reserved words**: cannot be used as a name

    E.g.: In C, you cannot have a variable named for, as it is a reserved word.

  Having reserved words reduces confusion. But if you end up having too many reserved words, it may become annoying for users, as they cannot use these words as variable, function, class, etc. names. For instance, COBOL has too many reserved words, including words that may be commonly used as user-defined names, such as START, STOP, STATUS.

- Some languages have *conventions for naming*: non-enforced style for naming things

  E.g.: In Java, by convention, class names are CamelCase and start with an uppercase letter. Methods are camelCase as well, but start with a lowercase letter.

Variables
Remember the Von-Neuman architecture: Variables are an abstraction of computer memory cells.

Attributes of variables:

- name
- address: machine memory address of the variable
- value: contents of the memory associated with the variable
- type: range of values the variable can have
- scope: range of statements where the variable is visible

Binding
Association between an entity and an attribute.

E.g.: variable ↔ name/address/type/value

Binding time: when the binding takes place. Some possibilities:

- language design time: e.g., + operator is bound to the summation operation
- compile time: e.g., a variable is bound to its type final int x = 5;
- link time: e.g., an extern function call is bound to the function definition
- run time: e.g., a variable is bound to its value

operators are hard coded. However, can be changed by operator overloading. (C++)

We will study binding of attributes to variables, which fall into two broad categories:

- *static* – occurs before runtime begins and remains unchanged
- *dynamic* – can change in the course of program execution

Type Binding

- static
  - explicit:
    E.g.: C++

    ```
    int n;
    Account acc;
    ```

  - implicit:
    E.g.: C#

    type of var is evaluated during compilation

    ```
    var sum = 0;
    var total = 0.0;
    var name = "Fred";
    var n = foo();
    ```

- dynamic
  E.g.: Python

  ```
  x = [ 10.2, 3.5 ]
  x = 47
  x = "abc"
  ```

  type of x can be changed on-the-fly.

Storage Binding & Lifetime

- Storage binding is created/destroyed with allocation/deallocation of memoryfor the variable
- Lifetime is the time during which the storage binding is active

Static variables: Bound to memory before execution begins and remains bound to the same memory cells throughout the execution. E.g.: static and global variables in the C language.

E.g.: C

```
int a = 5;            Global variable
int foo() {
    static int b;     Static variable
}
```

has fixed memory address

Static variables are stored in the *data segment*.

- Advantage: efficiency (no indirect addressing)
- Disadvantage: reduces flexibility, recursion is not possible

Layout of a C program:

BSS: Block Started by Symbol: typically includes all uninitialized objects declared at file scope; as well as uninitialized static local variables declared with the static keyword.

Cmd args & environment
↓ stack heap ↑
uninitialized data (BSS)
initialized data
program text

Static variables are allocated from here

BSS segment is typically 0-initialized. Since values of variables stored in the BSS segment need not be written into a binary image, BSS segment helps save space for program binaries (think about a large static array used by the program).

Stack-dynamic variables: Bindings are created when their declaration statements are elaborated (typically happens when the execution reaches the code that the declaration is attached to). E.g.: local variables and parameters of a function.

The storage binding of a typical stack-dynamic variable associated with a function starts when the function is called, and ends when the function returns.

E.g.: C

```
int foo(int x) {
    int y = x + 5;          Here, x and y are stack-dynamic
}
```

Stack dynamic variable are placed on the stack.

- Advantages: Makes recursion possible
- Disadvantages: Indirect addressing due to changing locations, allocation/deallocation overhead

Explicit heap-dynamic variables: Allocation and deallocation happens via explicit program instructions.

The storage binding starts when the explicit program instruction used for allocation (such as 'new' in C++) is executed and ends when the one used for deallocation (such as 'delete' in C++) is executed.

Explicit heap-dynamic variables are placed on the heap.

E.g.: C++

```
void foo() {
    int * intnode;
    intnode = new int;
    …                       Lifetime is between new and delete
    delete intnode;
}
```

It is important to make a distinction between the pointer and the pointed. In the above example *intnode (the pointed) is explicit heap-dynamic, but intnode (the pointer) is stack-dynamic.

- Advantages: Used for creating dynamic structures (list, trees, etc.)
- Disadvantages: allocation/deallocation overhead (higher than stack dynamic variables, think about heap fragmentation from your OS course), indirect addressing

Implicit heap-dynamic variables: Bound to heap storage when assigned values.

E.g.: In Python:

```
highs = [74, 84, 15]
```

To understand this, consider what this may translate into in Java:

```
Object highs = new ArrayList({74, 84, 15});
```

Consider the following C program:

```
int a1[10];
static int a2 = 5;

void foo() {
        static int a3[10];
        int a4[10];
        int * a5 = (int *) malloc(10*sizeof(int));
        …
        free(a5);
}
```

→ not initialized ; will stay at BSS until initialization.

Variable	Storage Location	Storage Lifetime	Scope (where it is visible)
a1	data segment (bss)	entire program	entire file
a2	data segment (initialized)	entire program	entire file
a3	data segment (bss)	entire program	foo function
a4	stack	foo call	foo function
a5	stack	foo call	foo function
*a5	heap	malloc to free	N/A → I wouldn't call this N/A

<u>Scope</u>
Range of statements in which the variable is visible. There are tow kinds: static vs dynamic.

Static scoping: Can be determined by just examining the code at compile-time. Specifically, a variable is typically visible within the code block it is defined.

→ python parser analyses the scope first to check if a variable with the same name was declared or not.

<table>
<tr><th></th><th>x</th><th>sub1</th><th>sub2</th><th>sub1-x</th><th>sub2-y</th></tr>
<tr><td>function big() {   but it is accessible in sub2() ?!!</td><td>|</td><td>|</td><td>|</td><td></td><td></td></tr>
<tr><td> function sub1() {   → x is not vis.</td><td>|</td><td>|</td><td>|</td><td>|</td><td></td></tr>
<tr><td>   var x = 7;   → x = 7</td><td>|</td><td>|</td><td>|</td><td>|</td><td></td></tr>
<tr><td>   x = x + 1   → x=8</td><td>|</td><td>|</td><td>|</td><td>|</td><td></td></tr>
<tr><td>   sub2();</td><td>|</td><td>|</td><td>|</td><td>|</td><td></td></tr>
<tr><td> }</td><td>|</td><td>|</td><td>|</td><td>|</td><td></td></tr>
<tr><td> function sub2() {   → x = 3</td><td>|</td><td>|</td><td>|</td><td></td><td>|</td></tr>
<tr><td>   var y = x;   → y = 3</td><td>|</td><td>|</td><td>|</td><td></td><td>|</td></tr>
<tr><td> }</td><td>|</td><td>|</td><td>|</td><td></td><td>|</td></tr>
<tr><td> var x = 3;   → x =3</td><td>|</td><td>|</td><td>|</td><td></td><td></td></tr>
<tr><td> sub1();   → x = 3</td><td>|</td><td>|</td><td>|</td><td></td><td></td></tr>
<tr><td>}</td><td>|</td><td>|</td><td>|</td><td></td><td></td></tr>
</table>

→ this example depends on the programming lang.

If there are more than one variable with the same name in the scope, then we have *shadowing*. In such cases, the variable from the block that is closest in the containment hierarchy will be used (as in the case of x = x + 1).

Blocks create new scopes as well:

```
…
if (…) {
        int tmp;
        ...
}
```

```
…
while (…) {
        int count;
        …
}
```

Some languages, such as C89, do not allow variable declarations that are not at the beginning of a block.

```
void foo() {
        int x = 0;
        int y = 5;
        …
}
```

```
void foo() {
        int x = 0;
        x = x+5;
        …
        int y = 5;
        …
}
```

Dynamic Scope

- Variable visibility is based on calling sequence
- You pick the variable from the closest scope in the calling sequence

```
function big() {
        function sub1() {
                var x = 7;
                sub2();
        }  ——> This might be wron
        function sub2() {
                var y = x; // which x?
                var z = 3;
        }  ——> should be here to encapsulate sub2()
        var x = 3;
        sub1();
}

big → sub1 → sub2
 x      x
```

```
function big() {
        function sub1() {
                var x = 7;
                sub2();
        }
        function sub2() {
                var y = x; // which x?
                var z = 3;
        }
        var x = 3;
        sub2();
}

big → sub2
 x
```

We take the x from sub1	We take x from big