Naming

Overview

- A critical ability of programming languages.
  - variables, functions, and programs need names before defining them.
- A name implies a binding, which is the connection between a definable entity and a symbol.
  - A binding is static if it takes place before run time.
    - Example: direct C function call. The function referenced by the identifier cannot change at run time
  - A binding is dynamic if it takes place during run time. (Compiled languages always dynamic)
    - Example: non-static methods in Java (methods that can be overridden or inherited). The specific type of a polymorphic object is not known before run time (in general), the executed function is dynamically bound.
- Names (identifiers) are determined by lexical rules.
- Case sensitivity

Case-insensitive	SQL, HTML, Pascal, Ada
Case-Sensitive	C, Java, Python
Mixed case sensitivity	PHP

  - In PHP, variables are case sensitive, but functions are not.

- Use of special characters: some languages allow use special characters

	Allowed	Not Recommend
Underscore (_)	C, Java, Python	SQL (considered as alias)

- Predefined identifiers
  - Reserved: keywords/reserved words
  - Most of PLs do not allow redefine the keywords. This is because it can make the parsing more efficiently.
    - For example, when parsing a program, the keyword, if, is always the start of a conditional statement. But if the keyword is redefined, it would need a second or more round to decide the meaning of the keyword.

Variables

- A variable represents the binding of an identifier to a memory address.
- Variables also have attributes such as their type, value, lifetime, and scope.
- The binding of a variable and address requires four pieces of information:
  - Identifier string
    - format of identifier stings 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.
  - The date 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

- Variables provide 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 reference.
  - 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.
- Examples

```
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, l-value (address) of x converted to an r-value
z = &x;

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

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

// l-value of z, l-value of y converted to an r-value, change value of z
z = &y + 4;

// l-value of x, l-value of z converted to an r-value
x = (int)z
```

In C, an l-value is always on the stack. In Java, it could be either

- **Some languages do not permit** these kinds of **explicit value manipulation of variables**. However, all languages must be **explicit in their internal semantic representation of code's meaning** of an identifier should be used in a particular statement.
- The situation in **Java** is more **complex**. The semantic model for variables treats **some types** of variables **to be memory locations**, but treat **other** types **as references**.
  - Consider an example, **i = 3**. The semantic **meaning** of the statement **depends on** the **type of i**.
  - If **i** is an **int**, the meaning of i is that it is **a memory location big enough to hold an integer** and the **integer value 3 is written to that location**.
  - If **i** is an **Integer**, the meaning of i is that it is **a memory location big enough to hold an address**, and it is **assigned the address of an Integer object** with the value 3.
  - Note that **if i is an Integer**, Java is implicitly executing the statement **i = new Integer(3)**; **Java must implicitly handle the difference between primitive and Object type**, because it **does not contain operators to manipulate the interpretation of variable**, as provided in C.

Scope vs Lifetime

- The scope of a declaration is the part of program for which the declaration is in effect.
  - 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: "visible" within function or statement block from point of declaration until the end of the block
  - Class scope: "seen" by class members
  - File scope: visible within the current file (a global variable in a C file has file scope)
  - Global scope: visible everywhere unless "hidden".

REF: https://www.csee.umbc.edu/~chang/cs202.f15/Lectures/modules/m05-scope/slides.php?print

- Lifetime of a variable is the time interval during which the variable has been allocated a block of memory.
  - Once the lifetime of a variable ends, the value the variable held is lost. Even if the variable 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 they lose their 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, Java, Python code below to demonstrate the scope in different languages.

```c
/**
 * staticInC.c
 * Ying Li
 * 02/23/2020
 */

#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 () {
     int i = 0;

     for (i = 0; i < 10; i++)
          foo();

     return 0;
}
```

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.

```c
/**
 * forLoopScopeInC.c
 * Ying Li
 * 02/23/20
 */

#include <stdio.h>

int gbl = 10; // program lifetime and file scope

int main () {
    int i = 1; // function scope

    printf("function i %d\n", i);

    if (1) { // 0 is false, non-zero is true
        int i = 2;
        printf("block i %d\n", i);

        // for loop in C has an implicit scope for loop variable declaration
        for(int i = 0; i < 5; i++, printf("for loop variable i %d\n", i)) {
            int i = 10;
            printf("for loop i %d\n", i);
        }

        // any for loop can be transformed to a while loop
        int j = 0;
        while (j < 5) {
            int i = 10;
            printf("while loop i %d\n", i);
            j++;
            printf("while loop variable j %d\n", j);

        }

        for (int i = 0; i < 3; i++) {
            printf("outer \n");
            for (int i = 0; i < 5; i++) {
                printf("inner ");
            }
            printf("\n");
        }
    }

    printf("function i %d\n", i);

    return 0;
}
```

Q1. What is the scope of the variable gbl and int i = 1? [gbl file scope, int i = 1 local/function scope]
Q2. Can the program be compiled successfully? [Yes]
Q3. What is the order of the two printf statements in the first for loop? Will the “for loop i” or the “for loop variable i” be printed first in each iteration? [“for loop i” is printed first]
For loop in C has an implicit scope for the loop variable declaration. So, C allows nested for loops share the name loop variable name.

```
##
#  scope.py
#
#  Ying Li
#  9/24/19
##

# no block scope for if statement
def demo1 ():
    if True:
        a = 1
    print(a)

# no block scope for if statement
def demo2 ():
    a = 0
    if True:
        a = 1
    print(a)

# no block scope for for loop
def demo3 ():
    for i in range(3):
        print("In loop ", i)
    print("After loop ", i)

# nested function scope
def outer ():
    def inner ():
        a = 0
        b = 0
        print("a in inner ", a)
        print("b in inner ", b)
    a = 1
    b = 2
    inner()
    print("a in outer ", a)
    print("b in outer ", b)

demo1()
demo2()
demo3()
outer()
```

```python
##
#  scope2.py
#
#  Ying Li
#  02/23/20
##

#g = 10

def foo ():
    global g
    g = 2

    print("g is ", g)

    for i in range(2):
        print i

    print("after for loop, i is ", i)

    for i in range(10):
        print i

foo()
print("global g is ", g)
#print(i) # not work due to function scope
```

```java
/**
 * scope.java
 * Ying Li
 * 02/23/2020
 */

public class scope {
    int field; // class scope

    // prameter field has constructor scope
    public scope (int field) {
        this.field = field;
    }

    public static void main (String argv[]) {
        if (true) {
            int q = 5; // if block scope

            System.out.println("q is " + q);

            if (true) {
                int q = 1; // can't shadow variable in nest block scopes
            }
        }
        // q is unavailable

        for (int i = 0, j = 5; i < j; i++, j--) {
            // cannot declare int i = 10, j = 20; here, as for loop
            // has no declaration scope for the loop variables
            System.out.println("i = " + i + " j = " + j);
        }
        // i and j are unavailable outside of the for block
    }
}
```

```java
/**
 * ExampleForStaticVariable.java
 * Static class fields are bound to the class not to individual instance
 * Ying Li
 * 02/23/2020
 */

public class ExampleForStaticVariable {

     private static int counter = 0;

     public void increaseCounter () {
          counter++;
     }

     public int getCounter () {
          return counter;
     }

     public static void main (String[] args) {
          ExampleForStaticVariable example1 = new ExampleForStaticVariable();
          ExampleForStaticVariable example2 = new ExampleForStaticVariable();
          example1.increaseCounter();
          example2.increaseCounter();
          System.out.println("example1's counter: " + example1.getCounter());
          System.out.println("example2's counter: " + example2.getCounter());
     }
}
```

```
##
# count.py
# class variables shared by all instances
# Ying Li
# 02/23/2020
##

class Count:
  count = 0

  def __init__ (self):
    self.count = 0

  def increment (self):
    Count.count += 1;
    self.count += 1

  def display(self):
    print "count = %d, self.count = %d\n" % (Count.count, self.count)

def main ():
  inst1 = Count()
  inst2 = Count()

  inst1.increment()
  inst2.increment()

  inst1.display()
  inst2.display()

if __name__ == "__main__":
  main()
```

Header Files of C

- A file containing C declarations and macro definitions to be shared between several source files.
- Header files server two purposes
  - System header files declare the interfaces to parts of the operation system. You include them in your program to supply the definitions and declarations you need to invoke system calls and library. `#include <stdlib.h>`
  - Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them. `#include "cstk.h"`
- More efficient code
  - Including a header file produces the same results as copying the header file into each source file that needs it. Such copying would be time-consuming and error-prone.
  - With a header file, the related declaration appear in only one place. Easy to update.
  - The directive works by directing the C preprocessor to scan the specified file as an input before continuing with the rest of the current file.
- Included files are not limited to declarations and macro definitions; those are merely the typical use. Any fragment of a C program can be included from another file. However, an included file must consist of complete tokens. Comments and string literals which have not be closed by the end of an included file are invalid. To avoid confusion, it is best if header files contain only complete syntactic units — function declarations or definitions, type declarations, etc.
- Once-Only Header
  - If a header file happens to be included twice, the compiler will process its contents twice. This is very likely to cause an error. Even if it does not, it will certainly waste time.
  - The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this:

```
/* File foo.*/
#ifndef FILE_FOO
#define FILE_FOO

the entire file

#endif /* end of file foo*/
```

  - This construct is commonly known as a **wrapper #ifndef**. When the header is included again, the conditional will be false. The preprocessor will skip over the entire contents of the file, and compiler will not see it twice.
  - CPP optimize even further. It remembers when a header file has a wrapper '`#ifndef`'. If a subsequent '`#include`' specifies that header, and the macro in the '`ifndef`' is still defined, it does not bother to rescan the file at all.
  - `FILE_FOO` is called the *controlling macro* or *guard macro.*
- REF: https://gcc.gnu.org/onlinedocs/cpp/Header-Files.html

```c
//
//  playExternCallee.c
//
//
//  Created by Ying Li on 2/23/20.
//

#include "playExtern.h"

int gblA = 3; // global lifetime, file scope
int *gblPtr = &gblA;

void printExternVal () {
    printf("Callee gblA Address: %p \n", gblPtr);
    printf("Callee gblA %d\n", gblA);
}


//
//  playExtern.h
//
//
//  Created by Ying Li on 2/23/20.
//

#ifndef playExtern_h
#define playExtern_h

#include <stdio.h>

extern int gblA; // global scope
//int gblA; // not an appropriate way
void printExternVal ();

#endif /* playExtern_h */



//
//  playExternCaller.c
//
//
//  Created by Ying Li on 2/23/20.
//

#include "playExtern.h"

//int gblA; // not an appropriate way

int main () {
    printExternVal();
    gblA = 10;
    printf("Caller gblA Address %p\n", &gblA);
    printf("Caller gblA %d\n", gblA);

    return 0;
}
```

Overloading

- Use of the same symbol to describe different functionality.
- Apply to functions and operators.
  - Variables have scope rules, and the binding to which a variable identifier is linked is unique for statically scoped languages.
  - Two variable bindings cannot have the same identifier within the local scope, and the scoping rules cause a local identifier to shadow a non-local binding.
  - Languages that permit access to non-local variables with the same identifier require the use of a prefix to make its name unique.
- Overloading of operators
  - Orthogonality: make the operators independent of the data type
  - Python allows operator overloading. The **+** operator functions on base data types string, int, float, and list, and ∗ operator functions on the base types when multiplied by an int type value.
  - In C, it is not possible to overload operators beyond their basic functionality, or to overload functions at all. Functions that apply to multiple data types must use generic pointers, such as a `void *`, or a `union` structure to pass around information. Responsibility for handling types correctly is the responsibility of the programmer. Creating new data types like a Vector or a Matrix requires making functions to implement standard mathematical operators.
  - C++ allows operators to be overloaded. If you create a Vector class, you can define + and * to do the right thing.
  - Java does not permit operator overloading.
- Overloading methods
  - C++ and Java allow overloading methods based on different signatures.
    - **Signature** usually includes the function name, number, type, and order of the parameters.
  - A class can have numerous functions with the same name, as long as they have different argument list.
  - Two identically named functions cannot differ only in their return types. Ada is the only language to permit overloading based on return type.
    - The difficulty with differentiating functions by only return type is twofold.
      - A function can be called without making use of the return value. Then there is no way to determine from the code which of the functions is meant to be executed.
      - Even if a function is on the right-hand side of an assignment, the variable on the left side may not be the same type as the return value of the function. While this is not an issue in strongly typed language like Ada, it is an issue in Java and C where automatic casts are common and do not generally raise errors or warnings.
- Methods and variables share the same identifiers [show them the codes `sameName.*`]
  - Java allow variables and functions to share the same identifier. The way to differentiate them is a method must have parentheses attached to the end of the identifier.
  - C and Python cannot allow methods and variables share the same name because in those languages a method name can be used as a right-hand side expression without any parentheses.

```c
/**
 * sameName.c
 * Ying Li
 * 02/23/2020
 */

int a () {
     return 0;
}

int main () {
     int a;
     a = a();

     return 0;
}
```

```java
/**
 * sameName.java
 * Ying Li
 * 02/23/2020
 */

class sameName {

     protected int a;

     public int a () {
          return 0;
     }

     public void b () {
          a = a();
     }

     public static void main (String[] args) {
          sameName s = new sameName();
          s.b();
     }
}
```

```python
##
# sameName.py
# Ying Li
# 02/23/2020
##

def a () :
  return 0

def main ():
  a = a()

if __name__ == "__main__":
  main()
```

A Dynamic Memory Allocation Example for Project 3

Q: If the size of an array is unknown till runtime, how do you create an array in C?
A: Dynamic memory allocation (malloc); Programmers control the lifetime of this type of memory
Exp: A Dog struct including name and age. But the name and age of an instance is read from the command line.
!: Remember to free the allocated memory before the end of the program.

```c
//
//  DogStruct.c
//
//
//  Created by Ying Li on 2/23/20.
//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    char *name;
    int age;
} Dog;

Dog *create_dog (char *nm, int a) {
    Dog *dog = (Dog*)malloc(sizeof(Dog));
    dog->name = (char*)malloc(sizeof(*nm));

    strcpy(dog->name, nm);
    dog->age = a;

    return dog;
}

void remove_dog (Dog *dog) {
    free(dog->name);
    free(dog);

    return;
}

int main (int argc, char **argv) {

    if (argc <= 2) {
        printf("Usage: ./a.out age name\n");
        return 0;
    }

    Dog *dog = create_dog(argv[2], atoi(argv[1]));

    printf("%s is %d years old\n", dog->name, dog->age);

    remove_dog(dog);

    return 0;
}
```