Names, Binding, and Scope

Attributes of variables

Six attributes of variables:

- name (identifier)
  - Rules are fairly standard
  - Explicit vs. implicit vs. inferred
- address (l-value)
  - Aliases
- value (r-value)
- type – why types?
- lifetime - the time during which the variable is bound to (associated with) a particular address
- scope - the region of the program from which the variable is visible (can be accessed)

Are lifetime and scope the same?

Names

A name is a string of characters (a word) that represents a program entity (variable, type, subprogram, ...)

- What design issues exist for names?

A **reserved word** is a word that can not be a user-defined name

- `if`, `else` and `for` are reserved words in C++

A **keyword** is a word that has a special meaning, but that can be redefined by the programmer

- `INTEGER` and REAL are type names in FORTRAN, but can also be used as variable names

It is common for an author to use **keyword** to mean both keywords and reserved words.

Bindings

A **binding** is an association between two entities.

A **static** binding occurs before runtime and doesn't change during execution.

- bindings of values to constants in C
- bindings of function calls to function definitions in C

A **dynamic** binding occurs or changes at runtime. Examples:

- bindings of values to variables
- bindings of member function calls to virtual member function definitions in C++
- all bindings of messages (method calls) to methods in Java

Type binding

Static binding

- Usual approach and is used in C, C++, Java, Pascal, ...
- Can be done in several ways:
  - explicitly, through variable declarations
  - implicitly, through rules or conventions.

Dynamic type binding

- The type of a variable can change at runtime
- Often means that variables are not declared
  - Found mostly in older language (LISP, BASIC, ...) and scripting (Perl, TCL, ...) languages

How is Python typed?

Advantages of different type bindings

Errors determined at runtime vs. compile time

- With static binding type issues go away after the compilation phase
- With dynamic binding type errors have to be addressed (if at all) during execution
  - Code that is not a type error often requires an implicit type conversion
  - Code that isn't executed isn't checked

Flexibility vs. reliable

- Functionality generally addresses the different philosophy of the language
- Interpreters are generally better at implementing flexibility than compilers

Lifetimes

The **lifetime** of a variable is the time during which the variable is bound to a particular address

Allocation is binding a variable to a memory location (an address)

Deallocation is returning a memory cell to free memory after it is unbound from a variable

There are four different lifetimes for variables: **static**, **stack-dynamic**, **explicit-heap dynamic**, and **implicit-heap dynamic**

Static

Lifetime is entire program

- Globals and constants
- `static` local variables in C or C++

Access can be direct

- Other types of variables often require indirect access

Location determined before execution

- No time spent allocating/deallocating memory

Cannot support recursion

Allocated space is reserved for the entire program

Stack dynamic

Lifetime is execution of the block they are declare in

- Non-static local variables declared in functions and blocks

Location is dynamically bound

- Variables on the runtime stack

Allows for recursion

Slower, indirect accesses

Need to access the variable in table to find it's exact location

Explicit heap-dynamic

Explicit heap-dynamic variables are memory cells explicitly allocated from the heap

- Memory allocated using **new** in Java and C++
- Usually anonymous (unnamed)
- The memory is generally access only by references or pointers

The lifetime is allocation time until explicit deallocation or garbage collection

- In C++, the lifetime is from the time it is allocated (using **new**) until it is deallocated (using **delete**)

Implicit heap-dynamic

Variables for which storage isn't allocated until the variable is given a value

The storage associated with the variable can change with every assignment to it

The lifetime is from one such assignment to the next

- strings and arrays (hashes) in Perl, arrays in JavaScript

Implicit heap-dynamic variables are extremely flexible, but also expensive to use

Garbage Collection

What is **garbage collection**?

When does memory allocated in C get reclaimed by the system?

When does memory allocated in Java get reclaimed by the system?

Scope

The **scope** of a variable is the region of the program text in which the variable is visible.

A variable is visible if it can be referenced (used, referred to, ...)

A variable is **local** if it is declared within a block of code

A variable that is visible in a block but not local to it is a nonlocal variable of that block

Scope can be static or dynamic

Static scoping

Almost all modern programming languages use static scoping

- Generally it is more efficient and easier to understand

Names (variable references) are bound to declarations statically (at compile time)

- This can be done using only the program text
- The declaration of the name in the closest enclosing block of the program is used

Interesting behavior occurs if blocks (and their associated scopes) can be nested

Blocks

Can variables be created in any block?

Where do we most commonly see “internal” variables in Java?

Another scoping level

- Stack based variables – they have very limited scope

In general a declaration for a variable hides a variable with same name but in a “larger” scope

- Java doesn’t allow variables declared in a control structure to have the same name as other variables
- Java does allow local variables to have the same name as instance variables

Dynamic scoping

A variable is bound to the most recently seen declaration of that name (at runtime)

The lifespan is determined statically

Access to a variable tends to be global, regardless of where the variable was created

- Access to a variable to determined by searching down the runtime stack until a declaration of the name is found
- Type-checking must be dynamic

Declaration order

Some languages require all declarations to appear at the beginning of the procedure/function

Many languages allow variable declarations to appear anywhere a statement can appear

How does the compiler deal with variables that may or may not be needed?

Global scope

Variables existing outside of the scope of any particular function/procedure

Advantages?

Disadvantages?

Do we see this in any modern language?

Named constants

Most modern languages provide named constants

C

```
#define PI 3.14159
```

Not really a named constant

C++

```
const double PI = 3.14159;
```

variables can be used when initializing constants

Java

```
final double PI = 3.14159;
```

Variables can only be assigned a value once but not necessarily when declared