Name, Scope, Binding

UNIT-1

Name

- *Name* - character string used to represent something else. They are usually identifiers, but operators (+, &, *) are also names.
- They allow us to refer entities in a program by a symbol instead of an address.
- They provide a level of abstraction in a program: classes for data abstraction, functions for control abstraction.
- They give us a better focus on some aspects of a program by reducing the conceptual complexity of the code.

Binding

- *Binding* - the operation of associating two things, like a name and the entity it represents.
- Binding time - the moment when the binding is performed (compilation, execution, etc).
- *Early* binding - *late* binding. Refers to the binding time.
- *Static* binding - *dynamic* binding. Refers to compilation vs runtime.
- *Polymorphism* - allowing a name (function, variable) to be bound to more than one entity.
- *Alias* - multiple bindings for the same entity.

Early / Late Binding

- *Type*: early binding constrains the type of the variable. Late binding lets that be decided when a value is assigned.
- *Function*: function known at compilation time, or left to be matched when the call is being executed.
- *Value*: late binding waits until the value/data assigned to a variable is needed before evaluating or loading it.

Dynamic Binding

- The exact meaning of each identifier (variable/function) is determined when the instruction is executed based on context.
- Example in Lisp: function A makes reference to a "global" variable x.
- Function B declares a local variable x and then calls function A.
- Within that function call, x is the local variable from B.

Binding Time

- *Language design* - fundamental aspects of the language, built-in functions, keywords.
- *Language implementation* - details such as the size of each type, file representation, runtime exceptions.
- *Programming* - algorithms, design of data structures.
- *Compilation* - mapping between higher-level constructs and machine code, static data.
- *Linking* - between function calls and external entities and their actual code.
- *Load* - virtual addresses, dynamic libraries.
- *Runtime* - virtual functions, values to variables, many more.

Object Lifetime

- Object - any entity in the program. Variables, functions.
- *Object lifetime* - the period between the object creation and destruction.
- *Binding lifetime* - the period between the creation and destruction of the binding.
- Usually the binding lifetime is a subset of the object lifetime.
- *Dangling reference* - when the binding exists after the destruction of the object. Example: deleting a pointer but not making it NULL.
- *Leak memory* - when an object still exists but there is no binding to it. Example: making a pointer NULL without deleting it first. Solved by garbage collection.

Object Allocation

- Static objects - they have an absolute address that exists for the duration of the program.
- *Global* variables, static local variables, runtime tables, function space for languages that don't support recursion, constants.
- *Stack* objects - Last in, first out (LIFO). Function space for languages that support recursion,
- *Heap* objects - can be allocated and deallocated at arbitrary times. Dynamically allocated parts of linked data structures, dynamically resized objects.

Function Space

- The stack of function calls contains a frame for each function.
- One frame contains:
- arguments, return values
- local variables, elaboration-time constants
- temporaries: intermediate values produced in complex computations
- bookkeeping information: return address, reference to the calling frame, debugging information.

Heap Management

- There is usually a linked list - free list - of all the memory blocks not in use.
- When an allocation demand is made, the program searches the heap for a free block of at least the requested size.
- *First fit* - returning the first block that fits the request.
- *Best fit* - returning the smallest block that fits the request.
- *Worst fit* - the largest block, to avoid fragmenting the memory
- *Pool* - dividing the list into sublists by size.
- *Compact* - moving the allocating heaps closer together to create larger free blocks. When moving an object one needs to update all the references to it.

Scope and Rules

- *Scope of a binding* - the textual region of the program in which a binding is active.
- *Scope* - sometimes a region of a program of maximal size in which no binding changes scope.
- *Referencing environment* - the set of active bindings at any given point in the program execution.
- The scope of bindings is determined by binding rules, included in the description of the language.

Scope of a Binding

- Usually the scope of a binding is determined statically, meaning at compilation time.
- When a function is called that has a local variable, the binding between the variable name and the instance of the variable local to the call is created.
- Any previous bindings for that same variable name are deactivated in the process (or hidden).
- When the function call ends, the previous binding for the name is restored.

Static Scopes

- Static scope - when the scope of a binding is determined during compilation.
- Sometimes called lexical scope.
- Current binding - the matching declaration whose block most closely surrounds the point in the program where the name is mentioned.
- Global scope - some languages only support global variables (Basic)
- Local static scope - for languages that do not support recursion and for static variables in other.

Dynamic Scope

- Bindings that are defined at runtime. They depend on the order in which functions are called.

Dr. Uma

- Current binding - the one encountered the most recently during the execution and that has not yet been destroyed.
- Type checking for dynamic scoping is done at runtime.

Scope Implementation

- Static scope relies on a symbol table which is a map or dictionary. For each symbol it contains information about it.
- Dynamic scope uses an association list or a central reference table. An association list is a pair name/value.
- For dynamic scoping, it is a list of symbols (name) associated with the scope (value).
- When new declarations are made, they are pushed into the list (which works as a stack). When a scope ends, it is popped out of the list.
- The binding is determined by a linear search in the list starting from the most recent definition going backwards.