2. Names, Scopes, and Bindings

Binding, Lifetime, Static Scope, Encapsulation and Modules, Dynamic Scope
Binding

- Names
- Variables
- Bindings
- Binding time
- Language design issues
Names are used for “variables,” functions (methods, procedures), types, etc.

Names use some form of identifier.

Languages differ.

Examples
- Start with a letter, then a letter or a digit
- Also allow underline to connect words
- Also allow a few other characters (e.g., $)
- Allow anything that cannot be a number
  - In Lisp, *&&&&^& is OK, and so is $1000
Variables

- What is a *variable*?
- Not (just) a name
  - The thing to which a C++ pointer points need not have a name
  - Java objects are always anonymous
    - Named variables *refer to* (nameless) objects
  - Some locations have more than one name
- Not (just) a memory location
  - Where a value is stored can change under operating system control (e.g., virtual memory)
A binding is an association between two things

Examples:
- name ⇔ memory location
- name ⇔ value
- value ⇔ memory location
- name ⇔ type
- memory location ⇔ type

Bindings are generally set up by declarations or definitions
- For languages without declarations, context must be used
Binding Time

- **Static** *(early – before runtime)*
  - Language design time *(e.g., int)*
  - Language implementation time *(e.g., stack size)*
  - Program writing time *(e.g., data structures)*
  - Compile time *(e.g., memory layout for program)*
  - Link-load time *(e.g., memory addresses)*

- **Dynamic** *(late – during runtime)*
  - Runtime *(e.g., value of a variable)*
Some bind types to variables at compile time, some at run time
- In C, C++, Java, etc., “variables” are declared to be of a certain type
- In Python, type of a “variable” depends on use – each assignment defines the “variable”

Some bind function calls to code at compile time, some at run time
- In C, function call can be compiled to an address
- C++ and Java have virtual functions, where calls are bound at runtime
- Python functions are generic – even code changes at call
import java.util.Scanner;

public class FunctionBindings {
    static Scanner input = new Scanner(System.in);
    static class A {
        void f() {
            System.out.println("Using A version");
        }
        final void g() {
            System.out.println("Only one of these");
        }
    }
    static class B extends A {
        void f() {
            System.out.println("Using B version");
        }
    }
    public static void main(String[] args) {
        A x = new A();
        B y = new B();
        A z;
        int n = input.nextInt();
        if (n > 10) z = x; else z = y;
        z.f();
        z.g();
    }
}

Which f?
Depends on input
Only one of these
(final declaration tells compiler so)

OK – every B is also an A
Example using Python

```python
def combine(first, second):
    return first + second
```

Can’t be compiled – types of parameters not known until runtime

```python
In [2]: combine(2,3)
5
In [3]: combine(2,3.1)
5.1
In [4]: combine("two", "three")
'twothree'
In [5]: combine(2, "three")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```
Lifetime

- Basic concepts
- Lifetime of storage bindings
- Static allocation
- Automatic allocation
- Dynamic allocation
Basic Concepts

- **Lifetime** refers to *period of time* between creation and destruction.
- **Bindings** have lifetimes.
- **Note:** Lifetime of a storage binding can be considered the lifetime of the *object* stored.
- **Examples:**
  - Period during which a value is bound to a memory location.
  - Period during which a “local variable” exists.
Lifetime of Storage Bindings

- Binding of storage (memory) to address
- **Static**
  - Exists “forever” – throughout execution of program (i.e., life begins before runtime, and ends after runtime)
  - *Examples*: global variable, C/C++ static local variable
- **Automatic**
  - Exists during execution of a *code block*
  - Often called *stack binding*
  - *Example*: local variable in a function or block
Dynamic

- Creation under program control
- Destruction under program control (C++), or controlled by garbage collection (Java)
- Often called heap binding
- Example: object allocated by C++ new operator
  ```
  int * p = new int;
  *p (not p) is bound dynamically
  ```
- Example: object allocated by Java new operator
  ```
  Object obj = new Object();
  New object (not obj) is bound dynamically
  ```
Static Allocation

- All objects created “before” program runs
  - Can be any time before programmer’s code executes
  - May be “statement zero” or set up at load time
- Minimal runtime management required
- Fast
- Does not allow recursion
- Does not allow automatic reuse of memory
Memory Organization

- Laid out before execution begins
- Lifetime of object is entire execution of program
Design Choices

- Only method used in (original) Fortran and Cobol
- In C/C++, variables declared at file scope or as static in a function block or in a class
- In Java, static (class) variables
  - *Note:* class variables are not “completely” static
  - They live from the time the class is loaded until the time it is “unloaded”
Usually coupled with concept of a block
- A section of code that executes sequentially, with a definite beginning (**block entry**) and end (**block exit**) of execution
  - Examples: function body or other code between curly brackets in C++ or Java

Object may be allocated
- At block entry
- When object’s definition is executed

Object de-allocated at block exit
Automatic Allocation (continued)

- Requires more runtime management
  - Blocks can be nested
    - Example: \( f \) calls \( g \) calls \( h \)
  - Stack used
  - For this reason, method also called stack allocation

- Allows recursion – just another function call
- Memory can be reused when de-allocated
- Not as fast as pure static approach – must stop during runtime to allocate/de-allocate
  - And this requires runtime support
Memory Organization

- Each block has a frame or activation record on the stack
- Lifetime of object is from allocation to block exit
Language Choices

- Originated in Algol
- Used in Pascal, C, C++, Java, Ada, and many others
- Syntax
  - Block enclosed between `begin` and `end` (Algol/Pascal/Ada) or in curly brackets (C/C++/Java)
  - Same syntax used to group statements
    - *Example*: to execute several statements in a `while` loop in C/C++/Java, put them between `{` and `}`
      - This has nothing to do with allocation
Design Issues

- Different things should look different
- Using same syntax for two different concepts is a poor idea
  - In a block, allocation and de-allocation are needed
  - No need for this mechanism if only grouping is wanted
- PL/I used different notations for these two
  - DO – END for grouping
  - BEGIN – END for block
Need for grouping is a language deficiency
Better if language has fully bracketed notation

Example: Visual Basic

```
If x < 5 Then
    Let y = x + 1
    Print y
Else
    Let z = x - 1
    Print z
End If
```

"brackets"
Another approach: indentation counts

Example: Python

```python
if x < 5:
    y = x + 1
    print y
else:
    z = x - 1
    print z
...
```

Grouped because of indentation
Objects created upon explicit request

Objects stay alive
- Until explicitly de-allocated
- “Forever”
  - May be recycled by runtime system because no longer usable
    - garbage collection

Objects must be referenced indirectly
- Can’t name something once program is running
- Use pointers or references
  - No real difference – sometimes “pointers” used when there is special notation, “references” when not
Dynamic Allocation (continued)

- Requires significant runtime management
- Uses a heap
  - Unrelated to data structure used for priority queues
- Causes memory fragmentation
- Use of pointers can cause aliasing
- With explicit de-allocation, can have
  - Dangling references
  - Memory leaks
Memory Allocation

- Times of allocation and de-allocation not predictable
- Lifetime of object is program controlled
- Heap can become fragmented

Defragmentation needed to meet this request
Allocation
- A special keyword can be used
  - `new` in C++, Java, Visual Basic
- A special method can be used
  - Creation methods in Python (constructors), Eiffel and Smalltalk
  - Note: Java and C++ have special creation methods (constructors) and use keyword!

De-allocation
- `delete` in C++
- Not needed in Java or VB, but set reference to `null` (Java) or `Nothing` (VB) to “encourage” recycling
Pointers

- Language (such as C++) may allow
  - both dynamic and non-dynamic objects
  - pointers to non-dynamic objects
- Such languages usually require different notation to distinguish between
  - Identifier representing an object
  - Identifier representing pointer to an object
- Example: C++ uses * for pointers

```c
int x = 3
int * p = new int;
p = 3;
```
Pointers in Practice

- Smalltalk, Python
  - All objects dynamic (therefore anonymous)
  - Only pointers have names
  - No need for special notation
  - In this case, pointers called *references*
- Pascal – pointers *only* for dynamic objects
- Java
  - *Primitive types must be* static or automatic
  - “*Objects” must be dynamic*
  - No need for special notation (call them references)
Static Scope

- Basic concepts
- Approaches
- A different approach
- Static links
Name bindings have scope

“Scope” refers to region of program text in which binding applies

For now, look at static scope: binding determined by position in text

Note: Scope of a name binding is often thought of as the scope of the name

Example:
- Portion of text in which a variable name refers to a particular declaration
Approaches

- **Single global scope**
  - Each name extends over entire program text
  - Used in
    - Original BASIC
    - Most early COBOL programs

- **Global + local scopes**
  - Either throughout program or only in a single procedure
  - FORTRAN
    - Local by default
    - Use `COMMON` to declare global
      - But name can be different in different places!
Nested scopes
- Originated in Algol 60
- Each scope entirely contained in (nested in) another
- One binding per name per scope
- Done within blocks
- Closest nested scope rule

Limitations
- C/C++/Java allow nested blocks, but not nested functions
- Python and others allow nested functions as well
Example

- P2 and P4 are nested within P1
- P3 is nested within P2
- F1 is nested within P4
When a language does not declare variables, some other rule is required

Python
- Uses no declarations
- Uses Algol-style rule only when value is not changed
- Considers assignment to a variable a new definition
- For global variables only, and only if declared as such in a function, you may assign new values
Sample Program

```python
x = 1  # a global variable
y = 2  # another global variable

def f():
    print "In f"
    x = 12  # local
    y = 13  # local
    print "x = ", x
    print "y = ", y

def g():
    global y
    x = 22  # local
    y = 23  # global because of declaration
    print "In g"
    print "x = ", x
    print "y = ", y

# Main program
print "In main before using functions"
print "x = ", x
print "y = ", y
f()
print "In main after using f"
print "x = ", x
print "y = ", y
g()
print "In main after using g"
print "x = ", x
print "y = ", y
```

### In main before using functions
- x = 1
- y = 2

### In f
- x = 12
- y = 13

### In main after using f
- x = 1
- y = 2

### In g
- x = 22
- y = 23

### In main after using g
- x = 1
- y = 23
At runtime, scopes do not occur in order on stack

Need static links

Example

**Static structure:**
C, D nested in B, and B, E in A

**Dynamic structure:**
A calls E calls B calls D calls C
Encapsulation and Modules

- Inadequacy of block structure
- Idea of a module
- Module as manager
- Module as type
Sometimes 2 separate functions must share a variable

Local and global variables not adequate

- Static global variables are dangerous
- Automatic local variables
  - Have limited scope (can’t use binding outside function)
  - Have limited lifetime (no call-to-call “memory”)
- Static local variables
  - Have limited scope (can’t use binding outside function)
  - Have unlimited lifetime (call-to-call “memory”)

Inadequacy of Block Structure
Example: A Counter

Python

```python
# Function using a static variable
count = 0

def f():
    global count
    print "Entering f again. This is entry number ",
    count = count + 1
    print count

for i in range(0, 25):
    f()
```

C++

```c++
#include <iostream>
#include <string>
using namespace std;

// Function having a static local variable
void f()
{
    static int count = 0;
    cout " ""Entering f again. This is entry number 
"< count
    cout " " " count << endl;
}

int main()
{
    for(int i = 0; i < 25; ++i)
        f();
    return 0;
}```
Example: Stacks

- Functions
  - pop, push, top, empty
  - Must be accessible to user
  - Must have access to data

- Data
  - Array storage, integer topIndex
  - Must be hidden from user
  - Must be accessible to all functions

- If data variables are local to one function, they are not available to other functions
- If variables are global, they are not hidden from users
An abstract data type (ADT), containing
- Functions
- Data

Functions manage data
Data is accessible to all functions in the module
Data is hidden from all users of functions
Module *encapsulates* an ADT
Essentially, a module is a *name space* mechanism
New language construct, which manages an ADT

Module encapsulates an ADT, and contains
- Types – accessible, but details hidden
- Data for types – not accessible
- Functions operating on types of ADT – accessible

User must
- Import module
- Use encapsulated types to declare variables
- Use encapsulated functions on variables
Example in Modula 2

MODULE stack_manager;
IMPORT element, stack_size;
EXPORT stack, init_stack, push, pop;
TYPE
  stack_index = [1..stack_size];
  STACK = RECORD
    s : ARRAY stack_index OF element;
    top : stack_index; (* first unused slot *)
  END;

PROCEDURE push(VAR stk : stack; elem : element);
BEGIN
  IF stk.top = stack_size THEN
    error;
  ELSE
    stk.s[stk.top] := elem;
    stk.top := stk.top + 1;
  END;
END push;

PROCEDURE pop(VAR stk : stack) : element;
BEGIN
  IF stk.top = 1 THEN
    error;
  ELSE
    stk.top := stk.top - 1;
    return stk.s[stk.top];
  END;
END;

Only these are accessible to application

Application

var A, B : stack;
var x, y : element;
...
init_stack(A);
init_stack(B);
...
push(A, x);
...
y := pop(B);
You can’t make a stack manager
…or a stack
…or access parts of a stack

But you can create a stack
**reference** *(the class is public)*

And the functions are accessible
because they are public
And one of the functions makes you a stack, even though you
can’t use the constructor

```java
public class StackManager {
    private StackManager() {};

    public static class Stack {
        private static int SIZE = 100;
        private int[] storage;
        private int topIndex;

        private Stack() {}
    }

    public static Stack makeStack() {
        Stack s = new Stack();
        s.storage = new int[s.SIZE];
        s.topIndex = -1; //indicating empty stack
        return s;
    }

    public static boolean isEmpty(Stack s) {
        return s.topIndex == -1;
    }

    public static int top(Stack s) {
        return s.storage[s.topIndex];
    }

    public static void push(Stack s, int n) {
        ++s.topIndex;
        s.storage[s.topIndex] = n;
    }

    public static void pop(Stack s) {
        --s.topIndex;
    }
}
```
Because you can declare a Stack and make one using a public (and static) function, you can start.

Once you have a stack, you can use the public (and static) stack functions.

But this is awfully roundabout for Java.

```java
public static void main(String[] args) {
    StackManager.Stack a = StackManager.makeStack();
    System.out.println("Stack test:");
    for (int i = 0; i < 5; i++)
    {
        System.out.println("Pushing " + i);
        StackManager.push(a, i);
    }
    while (!StackManager.isEmpty(a))
    {
        int i = StackManager.top(a);
        System.out.println("Popping " + i);
        StackManager.pop(a);
    }
}
```
Module as Type

- New language construct, which is an ADT
- Module encapsulates an ADT, which contains a single type, and
  - Data for type – not accessible
  - Functions operating on type – accessible
- Since module encapsulates single type, use same name for both module and type
- User must
  - Use encapsulated type to declare variables
  - Use encapsulated functions on variables
Limitation to one type usually (not always) what is wanted

Two styles for use of encapsulated functions
- Parameter of encapsulated type treated as others
  \texttt{push(a, x);} \\
- Parameter of encapsulated type treated specially
  \texttt{a.push(x);} \\
  Function invocation may be thought of as \textit{sending message} to parameter of encapsulated type
No need for another type – stack manages itself

In manager approach
- data belongs to a data class
- functions belong to the manager class
Here, both data and functions are part of the one and only class

```java
public class Stack {
    private static int SIZE = 100;
    private int[] storage;
    private int topIndex;

    public Stack() {
        storage = new int[SIZE];
        topIndex = -1; // indicating empty stack
    }

    public boolean isEmpty() {
        return topIndex == -1;
    }

    public int top() {
        return storage[topIndex];
    }

    public void push(int n) {
        ++topIndex;
        storage[topIndex] = n;
    }

    public void pop() {
        --topIndex;
    }
}
```
Example using Java (continued)

- Note the difference from the user program for the stack manager
- The improvement is due to combining the stack and its manager
  - The stack “manages itself”
- Encapsulation is still key

```java
public class UseStack {
    public static void main(String[] args) {
        Stack a = new Stack();
        System.out.println("Stack test:");
        for (int i = 0; i < 5; i++) {
            System.out.println("Pushing " + i);
            a.push(i);
        }
        while (!a.isEmpty()) {
            int i = a.top();
            System.out.println("Popping " + i);
            a.pop();
        }
    }
}
```
Dynamic Scope

- Static scope vs. dynamic scope
- Example
- Difficulties
Static Scope vs. Dynamic Scope

- **Static (lexical) scope**
  - Binding in effect determined by *position in program text*
  - Scope of binding evident from *text* of program
  - Easy for compiler, hard for interpreter

- **Dynamic scope**
  - Binding in effect determined by *when encountered during execution*
  - Scope of binding determinable only by *execution* of program
  - Easy for interpreter, hard for compiler (which must simulate execution)
Using static scope, 1 is printed
Using dynamic scope 1 or 2 is printed, depending on input

Which $a$?
Using dynamic scope, the meaning of a function can be changed by the caller. Which max_score?