

CSC 533: Programming Languages

Spring 2012

Language features and issues

- variables & bindings
- data types
 - primitive
 - complex/structured
- expressions & assignments
- control structures
- subprograms

We will focus on C, C++, and Java as example languages

1

Variables

imperative programming languages tend to be abstractions of the underlying von Neumann architecture

variables correspond to memory cells

variable attributes

- name
- type – determines the range of values and operations
- address (memory location)
- value
- scope – range of statements in which the variable is accessible
- lifetime – time during which the variable is bound to an address

2

Static vs. dynamic

the time & manner in which variables are bound to attributes is key to the behavior/implementation of the language

- *static binding*: prior to run-time, remains fixed
 - usually more efficient
- *dynamic binding*: occurs or can change during run-time
 - usually more flexible

3

Binding name (What names can refer to variables?)

names are used to identify entities in programs

- *length* originally, 1 char names only
FORTRAN – 6 chars
COBOL – 30 chars
C – arbitrary length, only first 8 chars are significant
C++ & Java – arbitrary length, all chars significant (???)
- *connectors* C, C++, Java, COBOL, Ada all allow underscores
Scheme allows hyphens, ?, !, ...
- *case-sensitive* debate about desirability

studies have shown:

<code>maxScoreInClass</code>	<i>best</i>
<code>max_score_in_class</code>	<i>ok</i>
<code>maxscoreinclass</code>	<i>worst</i>

4

Binding name (cont.)

special words are words with preset meanings in the language

- *keyword*: has predefined meaning in context
can be used freely in other contexts, can be overridden

e.g., in FORTRAN

<code>REAL X</code>	declaration
<code>REAL = 3.5</code>	assignment

- *reserved word*: cannot be reused or overridden
in most languages (incl. C, C++ & Java), special words are reserved

In ALGOL 60, special words had to be written in a distinct font

5

Binding type (When is the type of a variable decided?)

static (compile-time)

- explicit declarations (most modern languages)

<code>num : integer;</code>	Pascal
<code>int num;</code>	C/C++/Java

- implicit declarations

In FORTRAN, if not declared then type is assumed

starts with I-N → INTEGER, else REAL

TRADEOFFS?

dynamic (run-time)

- variable is given type on assignment, can change

e.g., JavaScript, PHP, Python, Perl

ADVANTAGE: flexible – can write generic code

DISADVANTAGE: costly – type checking must be done during run
error-detection abilities are diminished

binding greatly determines implementation

static → compilation

dynamic → interpretation

6

Binding address (When is memory allocated & assigned?)

static: bound before execution, stays same

e.g., early versions of FORTRAN, constants, global variables in C/C++ & Java
static binding → no recursion **WHY?**

stack-dynamic: bound when declaration is reached, but type bound statically

e.g., could specify in FORTRAN 77, locals in C/C++, primitives & references in Java
can save space over static, but slower **WHY?**

heap-dynamic: bound during execution, changeable

- can be *explicit*: user allocates/deallocates objects
e.g., new/delete in C/C++, new in Java
efficient, but tricky (garbage collection helps)
- can be *implicit*: transparent to the user
e.g., JavaScript, Scheme

as with dynamic type binding: flexible, inefficient, error detection weakened

9

Binding address (cont.)

the three binding options correspond to common memory partitions

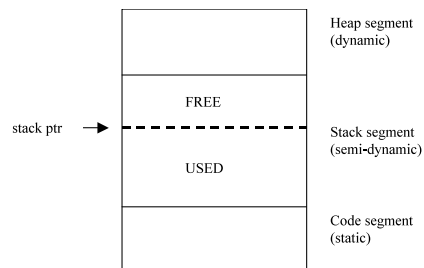
code segment: contains program instructions and static memory (note: compile-time)

stack-segment: contains stack-dynamic memory (note: run-time, LIFO)

- memory is allocated from one end (top)
- must be freed in reverse order (popped)

heap-segment: contains heap-dynamic memory (note: run-time, unpredictable)

- since lifetime is not predictable, must store as unstructured (linked) memory



boundary between the stack and heap can be flexible

- share same block of memory
 - grow from different ends
- advantages? disadvantages?

Note: LISP/Scheme is entirely heap based

10

Binding value (How are values assigned to variables?)

- consider the assignment: $x = y$
 - when on the left-hand side, variable refers to an address (l-value)
 - when on the right-hand side, variable refers to the value stored there (r-value)
- in C/C++/Java, an assignment returns the value being assigned

```
cout << x = 3 << endl;           System.out.println(x = 3);
```



```
x = y = 5;                        Note: assignment is right-associative
```
- some languages automatically initialize variables on declaration
 - e.g., JavaScript initialize to undefined
 - C/C++ initialize global primitives, Java initializes primitive fields
 - can specify initial values for user-defined types (class constructors)
- not all languages treat assignment the same

```
x = 2 + 3
```

In Prolog, right-hand side is not evaluated
X is assigned the operation '2+3'
can't change variables!

11

Scope & lifetime

lifetime is a temporal property; *scope* is a spatial property

- a variable is visible in a statement if it can be referenced there
- the scope of a variable is the range in which it is visible
- scope rules determine how variables are mapped to their attributes

scoping can be

- *static*: based on program structure (scope rules determined at compile-time)
- *dynamic*: based on calling sequence (must be determined during run-time)

12

Static vs. dynamic scoping

```
program MAIN;
  var a : integer;

  procedure P1;
  begin
    print a;
  end; {of P1}

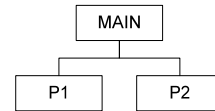
  procedure P2;
  var a : integer;
  begin
    a := 0;
    P1;
  end; {of P2}

begin
  a := 7;
  P2;
end. {of MAIN}
```

static (lexical)

non-local variables are bound based on program structure
if not local, go "out" a level

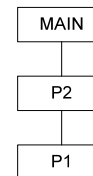
→ example prints 7



dynamic

non-local variables are bound based on calling sequence
if not local, go to calling point

→ example prints 0



13

Nested scopes

```
program MAIN;
  var a : integer;

  procedure P1(x : integer);
  procedure P3;
  begin
    print x, a;
  end; {of P3}
  begin
    P3;
  end; {of P1}

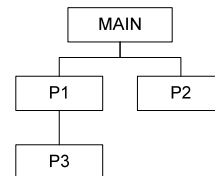
  procedure P2;
  var a : integer;
  begin
    a := 0;
    P1(a+1);
  end; {of P2}

begin
  a := 7;
  P2;
end. {of MAIN}
```

many languages allow nested procedures

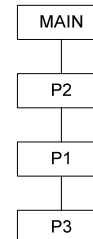
static scoping

→ example prints 1, 7



dynamic scoping

→ example prints 1, 0



14

In-class exercise

output using static scoping?

output using dynamic scoping?

```
program MAIN;
  var a : integer;

  procedure P1(x : integer);
    procedure P3;
      begin
        print x, a;
      end; {of P3}
    begin
      P3;
    end; {of P1}

  procedure P2;
    var a : integer;
    begin
      a := 0;
      P1(a+1);
    end; {of P2}

  begin
    a := 7;
    P1(10);
    P2;
  end. {of MAIN}
```

15

Nested scopes in C/C++/Java

in C/C++/Java, can't have nested functions/methods

but can nest blocks: new environments in {}

<pre>void foo() { int x = 3; if (x > 0) { int x = 5; cout << x << endl; } cout << x << endl; }</pre>	<pre>public void foo() { int x = 3; if (x > 0) { int x = 5; System.out.println(x); } System.out.println(x); }</pre>
---	--

Note: in C, variables can only be declared at the start of a block
in C++/Java, can declare variables anywhere

tradeoffs?

16

Scoping tradeoffs

virtually all modern languages are statically scoped

LISP was originally dynamic, as were APL and SNOBOL
Scheme is statically scoped, as is Common LISP (by default)
dynamic scoping is an option in Common LISP & Perl

serious drawbacks of dynamic scoping

- cannot understand a routine out of context
- cannot hide variables
- cannot type-check at compile time

static scoping is more intuitive & readable

- but can lead to lots of parameter passing
- also, moving code within a program can change its meaning

17

Static scoping and modularity

structured programming movement of the 70's (e.g., Pascal) stressed the independence of modules

- don't use global variables (and so don't rely on scope rules)
- result: all input to subprogram is clearly delineated as parameters

note: OOP methodology weakens modularity somewhat

- methods within a class have direct access to data fields
- is this desirable?
- avoidable?
- safe?
- note: optional "this." prefix helps

```
public class Die {
    private int numSides;
    private int numRolls;

    public Die() {
        this.numSides = 6;
        this.numRolls = 0;
    }

    public Die(int sides) {
        this.numSides = sides;
        this.numRolls = 0;
    }

    public int getNumberOfSides() {
        return this.numSides;
    }

    public int getNumberOfRolls() {
        return this.numRolls;
    }

    public int roll() {
        this.numRolls++;
        return (int) (Math.random()*this.numSides + 1);
    }
}
```

18