CSC 533: Organization of Programming Languages

Spring 2005

Language features and issues

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

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

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

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

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:

`maxScoreInClass`	*best*
`max_score_in_class`	*ok*
`maxscoreinclass`	*worst*

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

`REAL X`	declaration
`REAL = 3.5`	assignment

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

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

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

static *(compile-time)*

- explicit declarations (most modern languages)

`num : integer;`	Pascal
`int num;`	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, ISETL, APL, SNOBOL

  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

Binding type (cont.)

type checking: ensuring that operands in expressions are compatible

a *compatible* type is either
1. legal for that operator, or
2. can be coerced (automatically converted) into a legal type

coercion affects type checking, hence reliability

- most languages coerce numeric values
```
X = 3.5 + 2;
```
- C++ makes extensive use of coercion
```
char ch = 'A' + 1;

int * ptr = array[];
ptr++;
```
can define coercion paths for new classes

- Java allows C++-like coercion of primitive types
must explicitly convert reference types (objects)

Binding type (cont.)

a strongly typed language is one in which all type incompatibilities are caught (both compile-time & run-time)

- C++ somewhat weakly typed (mostly due to backward compatibility)
loopholes exist: = vs. ==
bool vs. int
union
unchecked parameters

- Java much better
closes many of C-style loopholes

Binding address (When is memory allocated & assigned?)

static: bound before execution, stays same

e.g., early versions of FORTRAN, constants, global variables in C++

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++, primitives 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++, 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

Binding address (cont.)

the three binding options correspond to common memory partitions

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

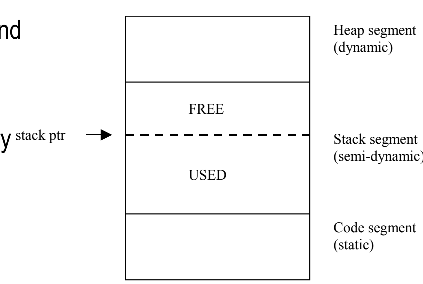
stack-segment: contains stack-dynamic memory (note: run-time, LIFO)

- 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

Note: LISP/Scheme is entirely heap based



boundary between the stack and heap can be flexible

- share same block of memory
- grow from different ends

advantages? disadvantages?

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++/Java, an assignment returns the value being assigned

```
cout << x = 3 << endl;

x = y = 5;
```

Note: assignment is right-associative

- Some languages automatically initialize variables on declaration
  e.g., JavaScript initialize to undefined
  C++ initializes 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!

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)

Static vs. dynamic scoping

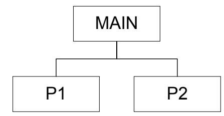
```
program MAIN;
  var a : integer;

  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

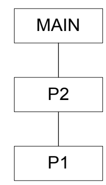


dynamic

non-local variables are bound
based on calling sequence

if not local, go to calling point

➔ example prints 0



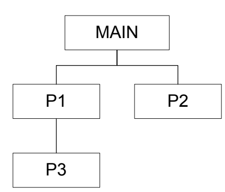
Nested scopes

```
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

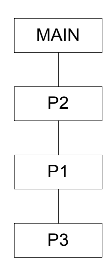
static scoping

➔ example prints 1, 7



dynamic scoping

➔ example prints 1, 0



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}
```


Nested scopes in C++/Java

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

but can nest blocks: new environments in { }

```
void foo()
{
  int x = 3;
  if (x > 0) {
    int x = 5;
    cout << x << endl;
  }
  cout << x << endl;
}
```

```
public void foo()
{
  int x = 3;
  if (x > 0) {
    int x = 5;
    System.out.println(x);
  }
  System.out.println(x);
}
```

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

tradeoffs?

Tradeoffs

virtually all modern languages are statically scoped

LISP was originally dynamic, as were APL and SNOBOL

later variants, Scheme and Common LISP, are statically scoped

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

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

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

member functions within a class have direct access to data fields

```
class foo {
  public:
    // MEMBER FUNCTIONS
  private:
    // DATA FIELDS
};
```

is this desirable? avoidable? safe?