

Basic Concepts: Variables, Bindings, Storage & Lifetime, Scoping and Types & Type Checking

CS216

1

Variables

CS216

2

A Programming Language – Universal: All Solvable Computations

- integer values and arithmetic operators (arithmetic expressions)
- **variables**
- assignment statement
- selection statement
- loop statement/go to statement

CS216

3

Variables

- An abstraction of a memory cell or collection of cells.
- Variables are more than just names for memory locations.

CS216

4

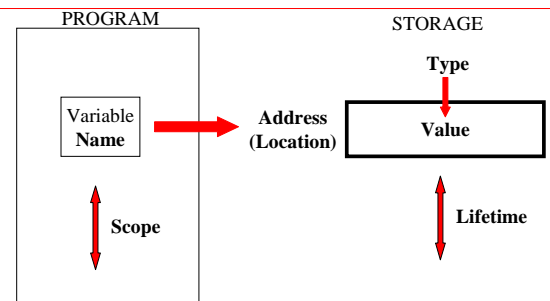
Attributes of Variables

- **Name**
- **Address**
- **Value**
- **Type**
- **Lifetime**
- **Scope**

CS216

5

Attributes of Variables



CS216

6

Variable Attribute - Name

- **Names** for variables are the most common names in a program.

CS216

7

Names/Identifiers

- A fundamental abstraction mechanism in a PL to denote entities or constructs.
- A string of characters used to identify some entity in a program.
 - Variables
 - Subprograms
 - Formal parameters
 - Others

CS216

8

Names/Identifiers – Design Issues

- Maximum length?
- Are connector characters allowed?
- Are names case sensitive?
- Are special words reserved words or keywords?

CS216

9

Names/Identifiers – Case Sensitivity

- Case sensitivity
 - C, C++, Java, and Modula-2 names are case sensitive
 - The names in other languages are not.
- Disadvantage?
 - Readability (names that look alike are different)
- Advantage?
 - ?

CS216

10

Names/Identifiers – Special Words

- A **keyword** is a word that is special only in certain contexts.
- A **reserved word** is a special word that cannot be used as a user-defined name.
- A **predefined word** has predefined meaning but can be redefined.

CS216

11

Variable Attribute – Address (Storage Location)

- The **address** of a variable is
 - The address of the memory location with which it is associated.
 - Called **l-value**
- The same name variable may have different addresses
 - At different places in a program.
 - At different times during execution.

CS216

12

Aliases

- Multiple names can reference the same address.
- If two variable names can be used to access the same memory location, they are called *aliases*.
- Aliases are harmful to readability!
 - Pointers, Pascal variant records, C and C++ unions, and FORTRAN EQUIVALENCE and through parameters.

CS216

13

Variable Attribute - Type

- The **type** of a variable is
 - (1) The range of values the variable can have.
 - (2) The set of operations that are defined for values of the type.

CS216

14

Variable Attribute - Value

- The **value** of a variable is
 - The contents of the location (memory cell) with which the variable is associated.
 - Called **r-value**.
- To access the r-value, the l-value is needed.
 - The *l-value* of a variable is its address.
 - The *r-value* of a variable is its value.

CS216

15

Variable Attribute – Lifetime

- The **lifetime** of a variable is
 - The time during which the variable is bound to a specific memory location.

CS216

16

Variable Attribute – Scope

- The **scope** of a variable is
 - The range of statements in which the variable can be referenced in the statements.

CS216

17

Binding

CS216

18

Binding

- An association between an attribute and an entity or between an operation and a symbol.
- **Binding time** is
 - The time at which a binding takes place.

CS216

19

Possible Binding Times

- Language design time
- Language implementation time
- Compile time
- Link time
- Load time
- Run (execution) time

CS216

20

Possible Binding Times

- Language design time
 - bind operator symbols to operations
- Language implementation time
 - bind floating point type to a representation
- Compile time
 - bind a variable to a type in C/C++ or Java
- Load time
 - bind a FORTRAN 77 variable to a memory cell (or a C static variable)
- Run (execution) time
 - bind a nonstatic local variable to a memory cell

CS216

21

Example: Possible Binding Times

```
int count;  
count = count + 5;
```

- Possible types for count?
 - At language design time
- Type of count?
 - At compile time
- Possible values of count?
 - At compile time
- Value of count?

CS216 At run time

22

Example: Possible Binding Times

```
int count;  
count = count + 5;
```

- Possible meanings of + ?
 - At language design time
- Meaning of + ?
 - At compile time
- Internal representation of 5?
 - At language implementation (compiler design) time

CS216

23

Static and Dynamic Binding

- **Static Binding**
 - Language design time
 - Language implementation time
 - Compile time
 - Link time
 - Load time
- **Dynamic Binding**
 - Run (execution) time

CS216

24

Binding Times

- **Early binding** times are associated with **greater efficiency!**
- **Later binding** times are associated with **greater flexibility!**

CS216

25

Storage (Location) & Lifetime

CS216

26

Storage Binding

- A memory cell to which a variable is bound is taken from a pool of available memory.
 - **Allocation**
 - Getting a memory cell from some pool of available cells.
 - **Deallocation**
 - Putting a memory cell back into the pool.

CS216

27

Lifetimes of Variables

- The **lifetime** of a variable is
 - The time during which it is bound to a particular memory cell.
 - From the allocation time to the deallocation time

CS216

28

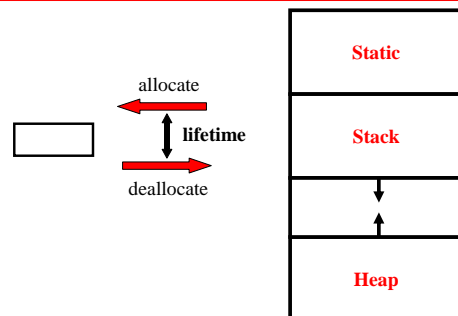
Three Kinds of Storage

- **Static storage**
 - Allocated and retained throughout the program's execution.
- **Stack storage**
 - Allocated and deallocated in last-in first-out order.
- **Heap storage**
 - Allocated and deallocated at arbitrary times.

CS216

29

Three Kinds of Storage



CS216

30

Categories of Variables

- **Static storage**
 - Static Variables
- **Stack storage**
 - Stack-Dynamic Variables
- **Heap storage**
 - Heap-Dynamic Variables

CS216

31

1. Static Variables

- Bound to a memory cell before the program's execution begins.
- Remains bound to the same memory cell throughout the program's execution.
 - All FORTRAN 77 variables, C static variables

CS216

32

Static Variables

- Advantage:
 - Efficiency (direct addressing)
 - **History-sensitive** subprogram support
- Disadvantage:
 - Lack of flexibility (no recursion)

CS216

33

2. Stack-dynamic Variables

- Storage bindings are created for variables when their declaration statements are elaborated.
 - Local variables in most languages

CS216

34

Stack-dynamic Variables

- Advantage:
 - Allows **recursion**
 - Conserves storage
- Disadvantages:
 - Overhead of allocation and deallocation
 - Subprograms cannot be history sensitive
 - Inefficient references (indirect addressing)

CS216

35

3. Heap-dynamic Variables

- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution.
- Referenced only through pointers or references
 - Dynamic objects in C++ (via new and delete), all objects in Java

CS216

36

Heap-dynamic Variables

- Advantage:
 - Provides for **dynamic storage management**
- Disadvantage:
 - Inefficient and unreliable

CS216

37

Scoping

CS216

38

Scopes of Variables

- A variable is **visible** in a statement
 - If it can be referenced in that statement.
- The **scope** of a variable is
 - The range of statements over which it is visible.

CS216

39

Two Occurrences of Variables

- Variables occur in two different contexts in a program
 - **Binding (defining, declaration) occurrence**
 - **Applied (use, reference) occurrence**

```
int x;  
  
x = x+3
```

CS216

40

The Scope Rule

- The **scope rule** of a programming language determines
 - How a particular occurrence of a name (variable) is associated with a variable.
- **Given an applied (use, reference) occurrence of a variable x, what is the binding (defining, declaration) occurrence of the variable x?**

CS216

41

Blocks - Program Building Blocks

- Blocks
 - Subprograms
 - functions and procedures
 - Non-subprogram blocks
 - begin ... end
 - { ... }

CS216

42

Program Structures

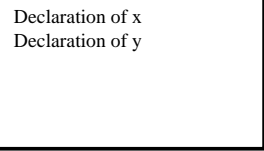
1. Monolithic block structured
2. Flat block structured
3. Nested block structured

CS216

43

1. Monolithic Block Structure

- Monolithic block structured
 - The entire program is a single block.



A single rectangular box representing the entire program. Inside the box, the text "Declaration of x" is on the top line and "Declaration of y" is on the bottom line.

CS216

44

Monolithic Block Structure

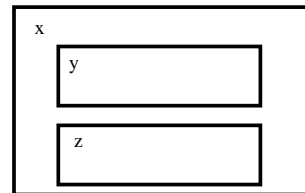
- All global variables.
- The scope of every declaration is the whole program.
 - All declared variables must have distinct names.
 - All declarations must be grouped even if they are used in different parts of the program.
 - Not suitable for writing large programs.

CS216

45

2. Flat Block Structure

- Flat block structured
 - The program is partitioned into two disjoint blocks.



CS216

46

Flat Block Structure

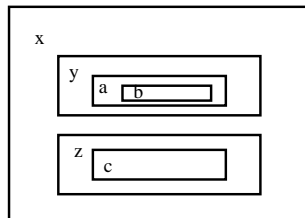
- Global or local variables.
- All subprograms and global variables must have distinct names.
- All variables that cannot be local to a particular subprogram is forced to be global even if it is accessed in a couple of subprograms.

CS216

47

3. Nested Block Structure

- Nested block structured
 - Each block may be nested inside any block.



CS216

48

Kinds of Variables

- Local variables
 - The **local** variables of a program unit or block are those that are declared there.
- Non-local variables
 - The **nonlocal** variables of a program unit or block are those that are visible but not declared there.
- Global variables
 - The **global** variables of a program unit or block are nonlocal variables that are declared in the outmost block.

CS216

49

Scope Binding

- Static binding
 - **Static scoping rule (Lexical scoping rule)**
- Dynamic binding
 - **Dynamic scoping rule**

CS216

50

1. Static Scoping

- Based on program text.
- Just by examining the program text, we can determine which binding occurrence correspond to a given applied occurrence.
- The binding between applied occurrences and binding occurrences is **FIXED**, not changing throughout the program's execution.

CS216

51

Static Scoping

- Search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name.
- **Find the innermost enclosing block containing the applied occurrence and a binding occurrence.**

CS216

52

Example: Static Scoping

- Example:
 - See the example (p. 212)
 - See the example (p. 213)

CS216

53

2. Dynamic Scoping

- Based on calling sequences of program units. (The program's dynamic flow of control)
- Not based on their textual layout (temporal versus spatial).
- The binding between applied occurrences and binding occurrences is changing throughout the program's execution.

CS216

54

Dynamic Scoping

- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.
- Find the most recently active block containing the applied occurrence and a binding occurrence.**

CS216

55

Example: Dynamic Scoping

- Example:
 - See the example (p. 217)

CS216

56

Static vs. Dynamic Scoping

- Programs in static scoped languages are
 - Easier to read
 - More reliable
 - Execute faster
- than equivalent programs in dynamic scoped languages.

CS216

57

Example: Static vs. Dynamic Scoping

```

MAIN
- declaration of x
SUB1
- declaration of x -
...
call SUB2
SUB2
...
- reference to x -
...
...
call SUB1
...
    
```

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

Static scoping - reference to x is to MAIN's x
 Dynamic scoping - reference to x is to SUB1's x

CS216

58

QUIZ: Static vs. Dynamic Scoping

<pre> Program example; const s = 2; var h: int; function scaled(d:int); return (d*s) procedure p; const s = 3; ... scaled(h); ... begin ... scaled(h); p; ... end </pre>	<div style="display: flex; justify-content: space-around;"> <div>Static Scoping</div> <div>Dynamic Scoping</div> </div>
<div style="display: flex; align-items: center;"> <div style="border: 1px solid black; padding: 2px; margin-right: 10px;">scaled(h);</div> <div style="flex-grow: 1; border-bottom: 2px solid red; position: relative;"> <div style="position: absolute; left: -10px; top: -5px;">→</div> </div> </div>	<div style="display: flex; justify-content: space-around;"> <div>2*h</div> <div>3*h</div> </div>
<div style="display: flex; align-items: center;"> <div style="border: 1px solid black; padding: 2px; margin-right: 10px;">scaled(h);</div> <div style="flex-grow: 1; border-bottom: 2px solid red; position: relative;"> <div style="position: absolute; left: -10px; top: -5px;">→</div> </div> </div>	<div style="display: flex; justify-content: space-around;"> <div>2*h</div> <div>2*h</div> </div>

59

Scopes vs. Lifetimes of Variables?

- Scope and lifetime are sometimes closely related.
- But, scope and lifetime are not the same and are different concepts!!**

CS216

60

Referencing Environment

- The **referencing environment** of a statement is
 - The collection of all names that are visible in the statement.

CS216

61

Types & Type Checking

CS216

62

Values and Types

- The **value** of a variable is
 - The contents of the location (memory cell) with which the variable is associated.
- The **type** of a variable is
 - (1) The range of values the variable can have.
 - (2) The set of operations that are defined for the values the variable can have.

CS216

63

Expressions and Types

- An expression is
 - A program phrase that will be evaluated to yield a value.
- An expression is used to compute a new value from an old value.
- The **type** of an expression is
 - (1) The range of values the expression can compute.
 - (2) The set of operations that are defined for the values the expression can compute.

CS216

64

Type Binding

- How is the type specified?
 - **Type declaration**
- When does the type binding take place?
 - **Static binding**
 - **Dynamic binding**

CS216

65

Type Declarations

- Explicit type declaration
- Implicit type declaration
 - No requirement for explicit type declaration

CS216

66

Type Inference

- The programmer does not need to specify the types of the variables.
- The types can be determined (inferred) from the context.
 - **Type Inferencing**
 - ML, Miranda, and Haskell

CS216

67

Example: Type Inference

```
fun circumf(r) = 3.14159 * r * r;  
fun times10(x) = 10 * x;  
fun square(x) = x * x;  
fun square(x):int = x * x;  
fun square(x:int) = x * x;  
fun square(x) = (x :int) * x;  
fun square(x) = x * (x :int);
```

CS216

68

1. Static Type Binding

- Type declaration creates static type binding.

CS216

69

2. Dynamic Type Binding

- A type is not specified by a type declaration.
- The variable is bound to a type when a value is assigned in an assignment statement.
- A type is specified through an assignment statement.

CS216

70

Dynamic Type Binding

- Advantage:
 - Flexibility (generic program units)
- Disadvantages:
 - High cost (dynamic type checking and interpretation)
 - Type error detection by the compiler is difficult

CS216

71

Type Binding and Implementation

- Languages with **static type binding** for variables are usually implemented by **compilation**.
- Languages with **dynamic type binding** for variables are usually implemented by **interpretation**.

CS216

72

Type Binding – Static vs. Dynamic Typing

- When does the type binding take place?
 - At compile time
 - Early type binding
 - **Static typing**
 - **Statically typed programming languages**
 - At run time
 - Late type binding
 - **Dynamic typing**
 - **Dynamically typed programming languages**

CS216

73

Type System

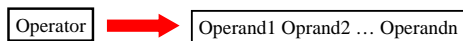
- A **type system** for a programming language is
 - A set of rules for associating a type with expressions in the language.
 - The rule of a type system specify the proper usage of each operator in the language.
- A type system rejects an expression if it does not associate a type with the expression.

CS216

74

Type Checking

- **Type checking** is
 - The activity of ensuring that the operands of an operator are of compatible types.
- Operator and its operands
 - Subprogram and its parameters
 - Assignment statement and its LHS (variable) & RHS (expression)



CS216

75

Type Compatibility

- A **compatible type** is one that is
 - either legal for the operator
 - or is allowed under language rules to be implicitly converted by compiler-generated code to a legal type.
 - **Type coercion.**

CS216

76

Type Error

- During execution, an error occurs if an operation is applied inappropriately.
- A **type error** is
 - The application of an operator to an operand of an inappropriate type.
- A program is **type safe** if it executes without any type errors.

CS216

77

Why Type Checking?

- Type errors account for a significant proportion of all program errors!
- To prevent type errors.

CS216

78

When Type Checking?

- Given an operator and its operands, the type checking must be performed **before** the operation is performed.
- Before when?
 - At compile time
 - Static type checking**
 - At run time (immediately before performing an operation)
 - Dynamic type checking**

CS216

79

Static vs. Dynamic Type Checking

- Static type checking
 - No time and space overhead - fast
 - More secure programs
- Dynamic type checking
 - Time and space overhead - slow
 - Flexible programs

CS216

80

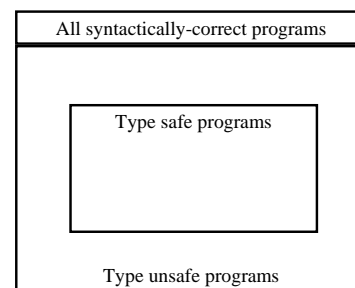
Type Binding and Checking Time

- If all type bindings are static, nearly all type checking can be done statically.
 - Static typing and static type checking?
 - Static typing and dynamic type checking?
- If type bindings are dynamic, type checking must be done dynamically.
 - Dynamic typing and static type checking?
 - Dynamic typing and dynamic type checking?

CS216

81

How Much Type Checking?

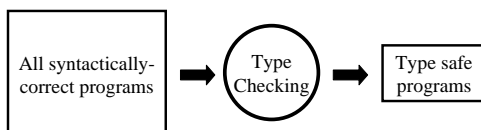


CS216

82

Strong Typing

- A programming language is **strongly typed**
 - If type errors are always detected.
 - If the type checking accepts only type safe programs.

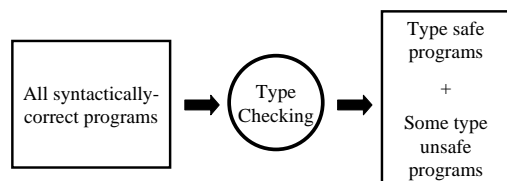


CS216

83

Weak Typing

- A programming language is **weakly typed**
 - If it is not strongly typed.



CS216

84

Strong Typing

- Strong typing ensures freedom from type errors!
- Languages:
 - FORTRAN 77 is not: parameters, EQUIVALENCE
 - Pascal is nearly strongly typed: variant records
 - Modula-2 is not: variant records.
 - C and C++ are not: parameter type checking can be avoided; unions
 - Ada and Java are nearly strongly typed.
 - ML is strongly typed.

CS216

85

Type Equivalence/Compatibility Rule

- What does it mean for two types are equal?
 - **Type Equivalence/Compatibility Rule**
- Rules:
 - **Name equivalence rule**
 - **Structural equivalence rule**
 - **Declaration equivalence rule**

CS216

86

1. Name Equivalence

- The two variables have compatible types if they are in
 - either the same type declaration
 - or in declarations that use the same type name.
- Easy to implement but highly restrictive.

CS216

87

Example: Name Equivalence

```
type
t1 = array [1..10] of integer;
t2 = t1;
t3 = t2;
t4 = array [1..10] of integer;
age = integer;
```

- $t1 \neq t2 \neq t3$
- $age \neq \text{integer}$
- $t1 \neq t4$

CS216

88

2. Structural Equivalence

- Two variables have compatible types if
 - their types have identical structures.
- More flexible, but harder to implement.

CS216

89

Example: Structural Equivalence

```
type
t1 = array [1..10] of integer;
t2 = t1;
t3 = t2;
t4 = array [1..10] of integer;
age = integer;
```

- $t1 = t2 = t3$
- $age = \text{integer}$
- $t1 = t4$

CS216

90

Structural Equivalence Issues

- Are two record types compatible if they are structurally the same but use different field names?
- Are two array types compatible if they are the same except that the subscripts are different? (e.g. [1..10] and [-5..4])
- Are two enumeration types compatible if their components are spelled differently?

CS216

91

3. Declaration Equivalence

- Type names that lead back to the same original structure declaration by a series of type redeclarations are considered to be equivalent types.

CS216

92

Example: Declaration Equivalence

```
type
  t1 = array [1..10] of integer;
  t2 = t1;
  t3 = t2;
  t4 = array [1..10] of integer;
  age = integer;
```

- $t1 = t2 = t3$
- $age = integer$
- $t1 \neq t4$

CS216

93

QUIZ: Type Equivalence

```
type
  range = -5 .. 5;
  t1 = array [range] of char;
  t2 = t1;
var
  x,y: array [-5..5] of char;
  z: t1;
  w: t2;
  i: range;
  j: -5..5;
```

Structural equivalence:
 $x = y = z = w$
 $i = j$

Name equivalence:
 $x = y$

Declaration equivalence:
 $x = y$
 $z = w$

CS216

94