

*"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."*

- Brian W. Kernighan

## CSE341 Programming Languages

Lecture 4 – November 2021

Data Types

© 2012-2021 Yakup Genç

Chapter 6 slides are adapted from R.W. Sebesta, C. Li & W. He and V. Shmatikov

## Scope

- The range of statements where a variable is visible, i.e. can be accessed
- Local variables are visible in the program unit where they are declared
- Non-local variables are visible in a program unit but not declared there
- Scope rules determine how names are associated with variables

November 2021

CSE341 Lecture 4

2

## Static Scope

- Also called Lexical Scope
  - The scope of variable can be determined statically (prior to execution)
- Based on structure of program text
- To connect a name to a variable, one (you or the compiler) must find the declaration
- Search process:
  - Search declarations, first locally,
  - Then in increasingly larger outer enclosing scopes
  - Until declaration for the name is found
- Enclosing static scopes (to a specific scope) are called its static ancestors
  - The nearest static ancestor is called a static parent

November 2021

CSE341 Lecture 4

3

## Shadowed Variables

- Variables are shadowed (hidden) in part of the code where there is a more immediate ancestor (closer in scope) with the same name
- C++ and Ada allow access to hidden variables using longer names
  - In Ada: <unit> .<name>
  - In C++: <class\_name> ::<name>
- Common Lisp
  - Packages variables accessible via longer name or package imported
- Java
  - this.<name>
  - <class\_name>.this.<name>
  - super.<name> // shadowed field in superclass

November 2021

CSE341 Lecture 4

4

## Scope Blocks

- Static scopes can be created inside program units
- Started with ALGOL 60!  
C and C++:  

```
for (int index; ...) {  
    int x;  
    ...  
}
```
- Scope block
  - statements in between {...}, and
  - condition expression in preceding or following (...)

November 2021

CSE341 Lecture 4

5

## Declaration Order

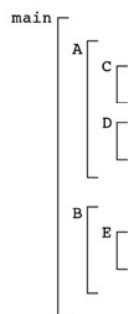
- Where should be variables declared
  - For some PLs, must appear at the beginning of functions
  - Some others, e.g., C++, Java, etc. can appear anywhere
- What scope
  - Only after declaration till the end of the block, e.g., in C++
  - Anywhere in the block, e.g., in C# and JavaScript
    - C# still requires declaration
    - JavaScript will result in “undefined”

November 2021

CSE341 Lecture 4

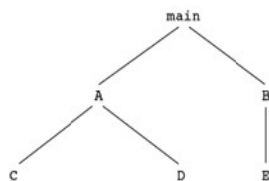
6

## Static Scoping Diagram



Example:

- MAIN calls A and B
- A calls C and D
- B calls A and E



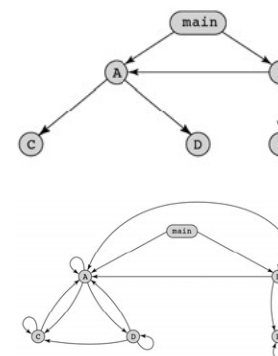
November 2021

CSE341 Lecture 4

7

## Static Scope Problems

- Suppose the specs are changed so that D must access data in B
- Solutions:
  - Put D in B
    - but then C can't call D and D can't access A's variables
  - Move the data from B that D needs to MAIN
    - but then all procedures can access them
- Same problem for procedure access
- Static scoping may encourage many global variables



November 2021

CSE341 Lecture 4

8

## Dynamic Scope

- Uses calling sequences of program units, not their textual layout
  - i.e. temporal versus spatial
- References are connected to declarations by searching back through the chain of subprogram calls in execution
  - i.e. search through the dynamic stack for the most recent variable with the name

November 2021

CSE341 Lecture 4

9

## Scope Example

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**

November 2021

CSE341 Lecture 4

10

## Static Scope Example

```
begin
  integer m, n;

  procedure hardy;
  begin
    print("in hardy -- n = ", n);
  end;

  procedure laurel(n: integer);
  begin
    print("in laurel -- m = ", m);
    print("in laurel -- n = ", n);
    hardy;
  end;

  m := 50;
  n := 100;
  print("in main program -- n = ", n);
  laurel(1);
  hardy;
end;
```

in main program -- n = 100  
in laurel -- m = 50  
in laurel -- n = 1  
in hardy -- n = 100  
/\* note that here hardy is called from laurel \*/  
in hardy -- n = 100  
/\* here hardy is called from the main program \*/

November 2021

CSE341 Lecture 4

11

## Dynamic Scope Example

```
begin
  integer m, n;

  procedure hardy;
  begin
    print("in hardy -- n = ", n);
  end;

  procedure laurel(n: integer);
  begin
    print("in laurel -- m = ", m);
    print("in laurel -- n = ", n);
    hardy;
  end;

  m := 50;
  n := 100;
  print("in main program -- n = ", n);
  laurel(1);
  hardy;
end;
```

in main program -- n = 100  
in laurel -- m = 50  
in laurel -- n = 1  
in hardy -- n = 1  
;; NOTE DIFFERENCE -- here hardy is called from laurel  
in hardy -- n = 100  
;; here hardy is called from the main program

November 2021

CSE341 Lecture 4

12

## Static Scope in Nested Loops

```

begin
  integer m, n;
  procedure laurel(n: integer);
    begin
      procedure hardy;
        begin
          print("in hardy -- n = ", n);
        end;

        print("in laurel -- m = ", m);
        print("in laurel -- n = ", n);
        hardy;
      end;

      m := 50;
      n := 100;
      print("in main program -- n = ", n);
      laurel(1);
      /* can't call hardy from the main program any more */
    end;
  end;

```

November 2021

CSE341 Lecture 4

13

## Scope Evaluation

- Advantage of dynamic scoping
  - convenience (no need to pass variables)
  - flexibility
- Disadvantage
  - poor readability (same name different non-local vars)
  - reliability (accessibility from multiple subprograms)
  - longer access times for non-local variables

November 2021

CSE341 Lecture 4

14

## Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a static variable in a C or C++ function
  - Scope is static and local to the function
  - Lifetime extends over the entire execution of the program

November 2021

CSE341 Lecture 4

15

## Referencing Environments

- Referencing environment (RE) of a statement
  - all names that are visible in the statement
- In statically-scoped language RE is
  - Local variables, and
  - All of the visible variables in all of the enclosing scopes
- In dynamically-scoped languages RE is
  - Local variables, and
  - All visible variables in all active subprograms
  - Active
    - execution of the subprogram has begun and
    - is not yet terminated

November 2021

CSE341 Lecture 4

16

## Named Constants

- A variable that is bound to a value only once
- Improved readability
  - E.g., “pi” instead of “3.14....”
- Parameterize a program
  - E.g., parameterizing length of an array
- C++ allows dynamic binding of values to named constants
  - “const int result = results \* width + 1;”
- C# allows static “const” as well as dynamic “readonly”

November 2021

CSE341 Lecture 4

17

## Initialization

- The binding of a variable to a value at the time it is bound to storage
- Static
  - Binding and initialization occur before run time
  - Initial value must be specified as a literal or an expression whose only non-literals are named constants
- Dynamic
  - Initialization is dynamic

November 2021

CSE341 Lecture 4

18

## Examples

- Consider the C preprocessor

```
#define ADD(x) x + a
```

November 2021

CSE341 Lecture 4

19

## Examples

- LISP calls dynamically scoped variables as special variables

```
> (defvar *special* 5)
*SPECIAL*
> (defun check-special () *special*)
CHECK-SPECIAL
> (check-special)
5
> (let ((*special* 6)) (check-special))
6
```

November 2021

CSE341 Lecture 4

20

## Examples

- Perl
- “my” for lexical scoping
  - A “my” variable has a block of code as its scope
  - A block is often declared with braces {}, but as far as Perl is concerned, a file is a block
  - A variable declared with “my” does not belong to any package, it 'belongs' only to its block
- “local” for dynamic scoping

November 2021

CSE341 Lecture 4

21

## Examples

```
first();
sub first {
    local $x = 1;
    my $y = 1;
    second();
}
sub second {
    print "x=", $x, "\n";
    print "y=", $y, "\n";
}
```

Local creates a new 'local' value for a global variable, which **persists** until the end of the enclosing block.

When control exits the block, the old value is restored.

But the variable, and its new 'local' value, are still global, and hence accessible to other subroutines that are called before the old value is restored.

second() therefore prints "x=1", because \$x is a global variable that temporarily happens to have the value 1.

November 2021

CSE341 Lecture 4

22

## Type

- A type is a collection of computable values that share some structural property
- A type is also a set of operations on the values
- Examples:
  - Integers, Strings,  $\text{int} \rightarrow \text{bool}$ ,  $(\text{int} \rightarrow \text{int}) \rightarrow \text{bool}$ , ...
- Not type examples:
  - {3, true}, , Even integers
- Distinction between sets that are types and sets that are not types is language-dependent

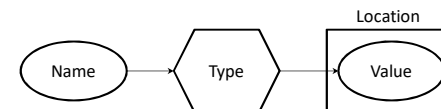
November 2021

CSE341 Lecture 4

23

## Variables Revisited

- Recall that variables have six attributes
  - Name
  - Type
  - Location or reference (l-value)
  - Value (r-value)
  - Scope - where variable accessible and modifiable – static vs. dynamic
  - Lifetime - interval of time in which location bound to variable



- An identifier (or name) is used to identify the entities, e.g., variable names, function names, etc., in a programming language

November 2021

CSE341 Lecture 4

24

## Variables Revisited

- A **descriptor** is the collection of the attributes of a variable
  - Implementation: an area of the memory where attributes of a variable is stored
    - All static attributes: descriptors are required only at compile time. Built by compiler and stored usually in symbol table and used during compilation.
    - If dynamic: Part or all off the attributes need to be maintained during execution. The descriptor is used by the run-time system.
  - Used for type checking and building the code for allocation and de-allocation

November 2021

CSE341 Lecture 4

25

## Uses for Types

- Program organization and documentation
  - Separate types for separate concepts
    - Represent concepts from problem domain
  - Indicate intended use of declared identifiers
    - Types can be checked, unlike program comments
- Identify and prevent errors
  - Compile-time or run-time checking can prevent meaningless computations such as `3+true` - “Bill”
- Support optimization
  - Example: short integers require fewer bits
  - Access record component by known offset

November 2021

CSE341 Lecture 4

26

## Uses for Types

- **Example: `z = x / y;` (Java)**
  - `int x, y; x=5; y=2;`
    - Integer division, `x/y` results in 2.
    - `int z: z = 2;`
    - `double z: z=2.0;`
  - `double x, y; x=5; y=2;`
    - floating-point division, `x/y` results in 2.5
    - `int z: wrong!`
    - `double z: z=2.5;`

November 2021

CSE341 Lecture 4

27

## Operations on Typed Values

- Often a type has operations defined on values of this type
  - Integers: `+` `-` `*` `<` `>` ...    Booleans: `^` `∨` `¬` ...
- Set of values is usually finite due to internal binary representation inside computer
  - 32-bit integers in C: `-2147483648` to `2147483647`
  - Addition and subtraction may overflow the finite range, so sometimes `a + (b + c) ≠ (a + b) + c`
  - Exceptions: unbounded fractions in Smalltalk, unbounded Integer type in Haskell
  - Floating point problems

November 2021

CSE341 Lecture 4

28

## Type Errors

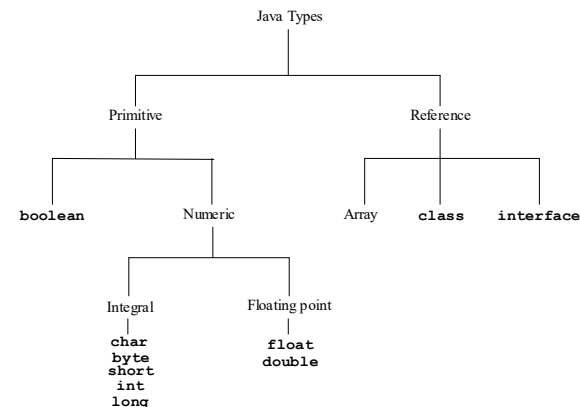
- Machine data carries no type information
  - 010000000101100000000000000000 means...
    - Floating point value 3.375?
    - 32-bit integer 1,079,508,992?
    - Two 16-bit integers 16472 and 0?
    - Four ASCII characters @ X NUL NUL?
- A type error is any error that arises because an operation is attempted on a value of a data type for which this operation is undefined
  - Historical note: in Fortran and Algol, all of the types were built in. If needed a type "color," could use integers, but what does it mean to multiply two colors?

November 2021

CSE341 Lecture 4

29

## Java Types

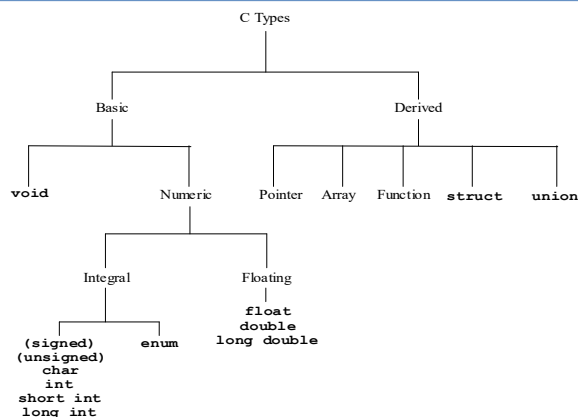


November 2021

CSE341 Lecture 4

30

## C Types



November 2021

CSE341 Lecture 4

31

## Simple Data Types

- No internal structure:
  - e.g., integer, double, character, and boolean
- Often directly supported in hardware
  - machine dependency
- Most predefined types are simple types
  - Exceptions: String in Java.
- Some simple types are not predefined
  - Enumerated types
  - Subrange types

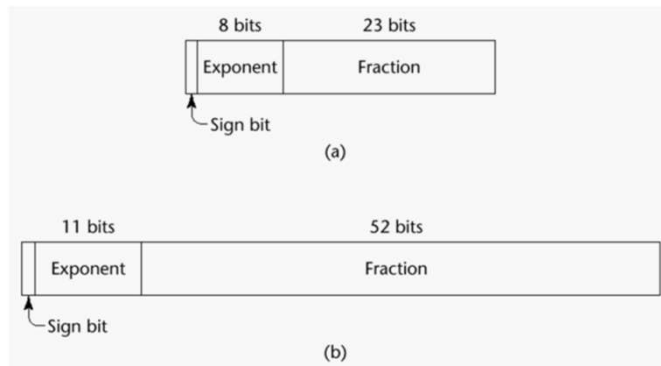
November 2021

CSE341 Lecture 4

32



## Floating Point



IEEE floating-point formats: (a) Single precision, (b) Double precision

November 2021

CSE341 Lecture 4

33

## Enumerated Types

**Ordered** set, whose elements are **named** and **listed** explicitly.

- Examples:

```
enum Color_Type {Red, Green, Blue};      ( C )
type Color_Type is (Red, Green, Blue);   ( Ada )
datatype Color_Type = Red | Green | Blue; ( ML )
```

- Operations:

Successor and predecessor

November 2021

CSE341 Lecture 4

34

## Ada Example

```
type Color_Type is (Red, Green, Blue);
```

```
x : Color_Type := Green;
```

```
x : Color_Type' Succ(x);
```

```
x : Color_Type' Pred(x);
```

```
put(x);      -- prints Green
```

- No assumptions about the internal representation of values
- Print the value name itself

November 2021

CSE341 Lecture 4

35

## Pascal Example

```
type
```

```
    cardsuit = (club, diamond, heart, spade);
```

```
    card = record
```

```
        suit: cardsuit;
```

```
        value: 1 .. 13;
```

```
    end;
```

```
var
```

```
    hand: array [ 1 .. 13 ] of card;
```

- Succ(diamond) = heart; Pred(spade) = heart;
- club < heart; is true.
- for acard := club to heart do

November 2021

CSE341 Lecture 4

36

## C Example

```
#include <stdio.h>
enum Color {Red, Green, Blue};
enum Courses {CSE1111=1, CSE3302=3, CSE3310=3, CSE5555=4};
main() {
    enum Color x = Green;
    enum Courses c = CSE3302;
    x++;
    printf("%d\n", x);
    printf("%d\n", Blue+1);
    printf("%d\n", c);
    return 0;
}
```

- enum in C is simply int
- Can customize the values

November 2021

CSE341 Lecture 4

37

## Java Example

```
public enum Planet { MERCURY (2.4397e6), EARTH (6.37814e6);
    private final double radius; // in meters
    Planet(double radius) {this.radius = radius; }
    private double radius() { return radius; }

    public static void main(String[] args) {
        for (Planet p : Planet.values())
            System.out.printf("The radius of %s is %f\n", p, p.radius());
    }
}
```

**java.util.Enumeration has different meaning**  
 for (Enumeration e = v.elements(); e.hasMoreElements(); )  
 System.out.println(e.nextElement());

November 2021

CSE341 Lecture 4

38

## Evaluation of Enumeration Types

- **Efficiency** – e.g., compiler can select and use a compact efficient representation (e.g., small integers)
- **Readability** -- e.g. no need to code a color as a number
- **Maintainability** – e.g., adding a new color doesn't require updating hard-coded constants.
- **Reliability** -- e.g. compiler can check operations and ranges of value.

November 2021

CSE341 Lecture 4

39

## C Example for Maintainability

```
enum Color {White, Green, Blue, Black};
enum Color {White, Yellow, Green, Blue, Black};
main() {
    enum Color x = Black;
    int i = x;
    while (i >= White){
        if (i < Green)
            printf("this is a light color!\n");
        i--;
    }
}

What if no enumeration?
if (i < 1) printf("this is a light color!\n");
Has to be changed to:
if (i < 2) printf("this is a light color!\n");
```

November 2021

CSE341 Lecture 4

40

## Ada Example for Reliability

```

type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
type DIRECTION is (NORTH, EAST, SOUTH, WEST);

GOAL : DIRECTION;
TODAY : DAY;
START : DAY;

TODAY := MON;
GOAL := WEST;
START := TODAY;

TODAY := WEST; -- Illegal: WEST is not a DAY value
TODAY := 5; -- Illegal: 5 is not a DAY value
TODAY := TODAY + START; -- Illegal: "+" is not defined for DAYS

```

November 2021

CSE341 Lecture 4

41

## Subrange Types

**Contiguous subsets** of simple types, with a *least* and *greatest* element.

- **Example:**  

```

type Digit_Type is range 0..9;           (Ada)
byte digit;    //-128..127
...
if (digit>9 || digit <0) throw new DigitException();

```
- **Not available in C, C++, Java. Need to use something like:**
- **Defined over ordinal types:**
  - ordered, every value has a next/previous element
    - E.g., integer, enumerations, and subrange itself

November 2021

CSE341 Lecture 4

42

## Type constructors: Defining New Types

- **Type constructors as set operations:**
  - Cartesian product
  - Union
  - Subset
  - Functions (Arrays)
- **Some type constructors do not correspond to set operations** (e.g., pointers)
- **Some set operators do not have corresponding type constructors** (e.g., intersection)

November 2021

CSE341 Lecture 4

43

## Cartesian Product

- **Ordered pairs of elements from U and V**  

$$U \times V = \{(u, v) \mid u \in U \text{ and } v \in V\}$$
- **Operations:**
  - **projection**  

$$p_1: U \times V \rightarrow U; \quad p_2: U \times V \rightarrow V$$

$$p_1((u,v))=u; \quad p_2((u,v))=v$$

November 2021

CSE341 Lecture 4

44

## Examples

struct in C

```
struct IntCharReal
{
    int i;
    char c;
    double r;
}
```

**int x char x double**

November 2021

CSE341 Lecture 4

45

## The same type?

```
struct IntCharReal
{
    int i;
    char c;
    double r;
}
```

```
struct IntCharReal
{
    int j;
    char ch;
    double d;
}
```

November 2021

CSE341 Lecture 4

46

## The same type?

```
struct IntCharReal
{
    int i;
    char c;
    double r;
}
```

```
struct IntCharReal
{
    char c;
    int i;
    double r;
}
```

November 2021

CSE341 Lecture 4

47

## Record/Structure not Exactly Cartesian Products

- Component selector: projection by component names

```
struct IntCharReal x;
x.i;
```

- Most languages consider component names to be part of the type
- Thus the previous two types can be considered different, even though they represent the same Cartesian product

November 2021

CSE341 Lecture 4

48

## ML: Pure Cartesian Product

```
type IntCharReal = int * char * real;
```

```
(2, #"a", 3.14)
```

```
#3(2, #"a", 3.14) = 3.14
```

November 2021

CSE341 Lecture 4

49

## Haskell

- **Example:**

```
data Shape =  
    Circle Float Float Float String |  
    Point Float Float String
```

Diagram: Arrows point from "Type Constructor" to "data Shape" and from "Data Constructors" to "Circle" and "Point".

- **Example uses...**

```
let c = Circle 0 0 10 "c"  
let p = Point 0 0 "p"
```

November 2021

CSE341 Lecture 4

50

## Haskell

- **Example:**

```
data Point a = Pt a a
```

- **Notes:**

Single constructor → tuple type (a Cartesian product of other types)

- **Above type is also polymorphic**

```
Pt 2.0 3.0      :: Point Float  
Pt 'a' 'b'     :: Point Char  
Pt True False  :: Point Bool  
Pt 1 'a'
```

November 2021

CSE341 Lecture 4

51

## Union

- $U \cup V = \{x \mid x \in U \text{ or } x \in V\}$

- data items with different types are stored in overlapping region, reduce memory allocation
- Only one type of value is valid at one time
- E.g.,

```
union IntOrReal {  
    int i;  
    double r;  
}
```

- **Different from records?**

November 2021

CSE341 Lecture 4

52

## C Style Unions

- C style unions allow the same bit pattern to be interpreted in multiple ways
- E.g., it's common to use a union to pack/unpack a fixed length binary blob into its constituent fields
- Useful in some systems programming contexts
- But it's a very low level feature and was excluded in most languages

November 2021

CSE341 Lecture 4

53

## Undiscriminated Union in C

```
union IntOrReal {
    int i;
    double r;
}
union IntOrReal x;
x.i = 1;
printf("%f\n", x.r);
```

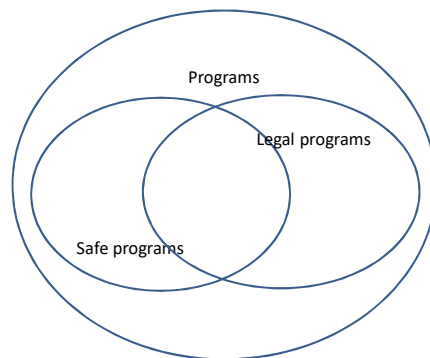
- Can be unsafe

November 2021

CSE341 Lecture 4

54

## Safe vs. Legal



November 2021

CSE341 Lecture 4

55

## Type Equivalence

- How to decide if two types are the same?
- Structural Equivalence
  - Types are sets of values
  - Two types are equivalent if they contain the same values
- Name Equivalence
  - Same type names
- Which one is more flexible?
  - Structural equivalence

November 2021

CSE341 Lecture 4

56

## Structural Equivalence of Type

- Two type expressions are structural equivalent if they are
  - same basic type, or
  - formed by applying the same constructor to two structurally equivalent types, or
  - after type declaration type  $n=T$  the type name  $n$  is structurally equivalent to  $T$
- two type expressions are structural equivalent if and only if they are identical
- Example:
  - `integer` is equivalent to `integer`
  - `pointer(char)` is equivalent to `pointer(char)`
- Modification is needed for structural equivalence: When array are passed as parameter, we may not wish to include the array bounds as part of the type

November 2021

CSE341 Lecture 4

57

## Structural Equivalence

```

struct RecA {
    char x;
    int y;
}
struct RecB {
    char x;
    int y;
}
struct RecC {
    char u;
    int v;
}
struct RecD {
    int y;
    char x;
}

```

Diagram illustrating structural equivalence:

- RecA and RecB are structurally equivalent (Char X Int).
- RecC and RecD are structurally equivalent (Int X Char).

November 2021

CSE341 Lecture 4

58

## But are they equivalent in these languages?

In C:

```

struct RecA {
    char x; int y;
};
struct RecB {
    char x; int y;
};
struct RecA a;
struct RecB b;

b = a;

```

Use structural equivalence for everything except unions and structs, which use name equivalence

( Error: incompatible types in assignment )

November 2021

CSE341 Lecture 4

59

## But are they equivalent in these languages?

In C:

```

struct RecA {
    char x; int y;
};
struct RecB {
    char x; int y;
};
struct RecA a;
struct RecB* b;

b = &a;

```

( Warning: incompatible types in assignment )

November 2021

CSE341 Lecture 4

60

## But are they equivalent in these languages?

In C:

```
struct RecA {
    char x;  int  y;
};
struct RecB {
    char x;  int  y;
};
struct RecA a;
struct RecB* b;

b=(struct RecB*) &a;  } ( OK, but does not mean they are equivalent )
```

November 2021

CSE341 Lecture 4

61

## But are they equivalent in these languages?

In Java:

```
class A {
    char x;  int  y;
};
class B {
    char x;  int  y;
};

A a = new B();
```

} Java enforces "name equivalency" – two types  $T_1$  and  $T_2$  are equivalent if and only if they have identical names.

November 2021

CSE341 Lecture 4

62

## Structural Equivalence Algorithm

```
boolean sequiv(s, t)
begin
    if s and t are the same basic type return true
    else if s = array( $s_1, s_2$ ) and t = array( $t_1, t_2$ ) then
        return sequiv( $s_1, t_1$ ) and sequiv( $s_2, t_2$ )
    else if s =  $s_1 \times s_2$  and t =  $t_1 \times t_2$  then
        return sequiv( $s_1, t_1$ ) and sequiv( $s_2, t_2$ )
    else if s = pointer( $s_1$ ) and t = pointer( $t_1$ ) then
        return sequiv( $s_1, t_1$ )
    else if s =  $s_1 \rightarrow s_2$  and t =  $t_1 \rightarrow t_2$  then
        return sequiv( $s_1, t_1$ ) and sequiv( $s_2, t_2$ )
    else return false
end
```

November 2021

CSE341 Lecture 4

63

## Equivalence Algorithm

If structural equivalence is applied:

```
struct RecA {
    char x;  int  y;
};
struct RecB {
    char u;  int  v;
};
struct RecA a;
struct RecB b;

b = a;
```

November 2021

CSE341 Lecture 4

64



## Replacing the names by declarations

```
typedef struct {
    char x;    int y;
} RecB;
RecB b;

struct {
    char x;    int y;
} c;
```

November 2021

CSE341 Lecture 4

65

## Replacing the names by declarations

```
typedef struct { char x; char y } SubRecA;
typedef struct { char x; char y } SubRecB;

struct RecA {
    int ID;  SubRecA content;
};

struct RecB {
    int ID;  SubRecB content;
};
```

November 2021

CSE341 Lecture 4

66

## Replacing the names by declarations?

```
typedef struct CharListNode* CharList;
typedef struct CharListNode2* CharList2;

struct CharListNode {
    char data;  CharList  next;
};

struct CharListNode2 {
    char data;  CharList2 next;
};
```

November 2021

CSE341 Lecture 4

67

## Cannot do that for recursive types

```
typedef struct CharListNode* CharList;
typedef struct CharListNode2* CharList2;

struct CharListNode {
    char data;  struct CharListNode* next;
};

struct CharListNode2 {
    char data;  struct CharListNode2* next;
};
```

**There are techniques for dealing with this**

November 2021

CSE341 Lecture 4

68

## Structural Equivalence

- Can be complicated when there are names, anonymous types, and recursive types
- Simpler, and more strict rules: name equivalence

November 2021

CSE341 Lecture 4

69

## Name Equivalence

```

struct RecA { char x; int y; };
typedef struct RecA RecB;
struct RecA *a;
RecB *b;
struct RecA c;
struct { char x; int y; } d;
struct { char x; int y; } e,f;

a=&c;      ( ok )
a=&d;      (Warning: incompatible pointer type)
b=&d;      (Warning: incompatible pointer type)
a=b;      ( ok. Typedef creates alias for existing name )
e=d;      ( error: incompatible types in assignment )

```

November 2021

CSE341 Lecture 4

70

## Type Equivalence in C

- Name Equivalence: struct, union
- Structural Equivalence: everything else
  - typedef doesn't create a new type

November 2021

CSE341 Lecture 4

71

## Example

```

struct A { char x; int y; };
struct B { char x; int y; };
struct { char x; int y; };
typedef struct A C;
typedef C* P;
typedef struct A * R;
typedef int S[10];
typedef int T[5];
typedef int Age;
typedef int (*F)(int);
typedef Age (*G)(Age);

struct A and C
struct A and B; B and C
struct A and struct { char x; int y; };
P and R
S and T
int and Age
F and G

```

November 2021

CSE341 Lecture 4

72

## Type Equivalence in Java

- No typedef: so less complicated
- class/interface: new type (name equivalence, class/interface names)
- arrays: structural equivalence

November 2021

CSE341 Lecture 4

73

## Example

```
long y;
float x;
double c;
x = y/2+c;
```

- $y$  long, 2 is int, so promoted to long,  $y/2$  long.
- $c$  is double,  $y/2$  is long, so promoted to double,  $y/2+c$  is double.
- $x$  is float,  $y/2+c$  is double, what happens?
  - C?
  - Java?

November 2021

CSE341 Lecture 4

74

## Example: C

```
struct RecA {int i; double r;};
int p( struct {int i;double r;} x)
{ ... }
int q( struct RecA x)
{ ... }

struct RecA a;
int b;

b = p(a);
b = q(a);
```

November 2021

CSE341 Lecture 4

75

## Type Conversion

- Use code to designate conversion?
  - No: automatic/implicit conversion
  - Yes: manual/explicit conversion
- Data representation changed?
  - No, just the type
  - Yes

November 2021

CSE341 Lecture 4

76

## Example: Java

- **Implicit conversion:**
  - Representation change (type promotion, e.g., int to double)
  - No representation change (upcasting)
- **Explicit conversion:**
  - Representation change (double x = 1.5; int y = (int)x)
  - No representation change (downcasting)

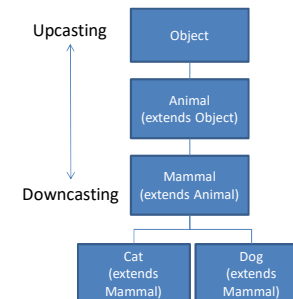
November 2021

CSE341 Lecture 4

77

## Upcasting and Downcasting

- Upcasting and downcasting allow building of complicated programs using simple syntax, and gives great advantages, like Polymorphism or grouping different objects
- **upcasting:** allowing an object of a subclass type to be treated as an object of any superclass type.
  - Upcasting is done automatically, while downcasting must be manually done by the programmer



November 2021

CSE341 Lecture 4

78

## Casting in Java

```

class A {public int x;}
class SubA extends A { public int y;}
A a1 = new A( );
A a2 = new A( );
SubA suba = new SubA( );
  
```

```

a1 = suba;           OK (upcasting)
suba = (SubA) a1;    OK (downcasting)
suba = a2;           compilation error
suba = (SubA) a2;    compiles OK, runtime error
a1.y;               compilation error
if (a1 instanceof SubA) { ((SubA) a1).y; }  OK
  
```

November 2021

CSE341 Lecture 4

79

## Type System

- **Type Constructors:**
  - Build new data types upon simple data types
- **Type Checking:** The translator checks if data types are used correctly.
  - **Type Inference:** Infer the type of an expression, whose data type is not given explicitly.  
e.g., x/y
  - **Type Equivalence:** Compare two types, decide if they are the same.  
e.g., x/y and z
  - **Type Compatibility:** Can we use a value of type A in a place that expects type B?  
Nontrivial with user-defined types and anonymous types

November 2021

CSE341 Lecture 4

80

## Type Checking vs. Type Inference

- Standard type checking

```
int f(int x) { return x+1; };
int g(int y) { return f(y+1)*2; };
```

- Look at the body of each function and use declared types of identifiers to check agreement

- Type inference

```
int f(int x) { return x+1; };
int g(int y) { return f(y+1)*2; };
```

- Look at the code without type information and figure out what types could have been declared

ML is designed to make type inference tractable

November 2021

CSE341 Lecture 4

slide 81

## Motivation

- Types and type checking

- Type systems have improved steadily since Algol 60
- Important for modularity, compilation, reliability

- Type inference

- Widely regarded as important language innovation
- ML type inference is an illustrative example of a **flow-insensitive static analysis algorithm**

November 2021

CSE341 Lecture 4

slide 82

## Type Inference Summary

- Type of expression computed, not declared
  - Does not require type declarations for variables
  - Find most general type by solving constraints
  - Leads to polymorphism
- Static type checking without type specifications
  - Idea can be applied to other program properties
- Sometimes provides better error detection than type checking
  - Type may indicate a programming error even if there is no type error

November 2021

CSE341 Lecture 4

slide 83

## Costs of Type Inference

- More difficult to identify program line that causes error
- ML requires different syntax for values of different types
  - integer: 3, real: 3.0
- Complications with assignment took years to work out

November 2021

CSE341 Lecture 4

slide 84

## Param. Polymorphism: ML vs. C++

- ML polymorphic function
  - Declaration has no type information
  - Type inference: type expression with variables, then substitute for variables as needed
- C++ function template
  - Declaration gives type of function argument, result
  - Place inside template to define type variables
  - Function application: type checker does instantiation

ML also has module system with explicit type parameters

November 2021

CSE341 Lecture 4

slide 85

## Example: Swap Two Values

- ML
 

```
fun swap(x,y) =
  let val t = !x in x := !y; y := t end;
val swap = fn : 'a ref * 'a ref -> unit
```
- C++
 

```
template <typename T>
void swap(T& x, T& y){
  T t = x; x=y; y=t;
}
```

Declarations look similar, but compiled very differently

November 2021

CSE341 Lecture 4

slide 86

## Implementation

- ML
  - Swap is compiled into one function
  - Typechecker determines how function can be used
- C++
  - Swap is compiled into linkable format
  - Linker duplicates code for each type of use
- Why the difference?
  - ML reference cell is passed by pointer, local t is a pointer to value on heap
  - C++ arguments passed by reference (pointer), but local t is on stack, size depends on type

November 2021

CSE341 Lecture 4

slide 87

## Another Example

- C++ polymorphic sort function
 

```
template <typename T>
void sort( int count, T * A[count] ) {
  for (int i=0; i<count-1; i++)
    for (int j=i+1; j<count-1; j++)
      if (A[j] < A[i]) swap(A[i],A[j]);
}
```
- What parts of implementation depend on type?
  - Indexing into array
  - Meaning and implementation of <

November 2021

CSE341 Lecture 4

slide 88

## Summary

---

- Types are important in modern languages
  - Organize and document the program, prevent errors, provide important information to compiler
- Type inference
  - Determine best type for an expression, based on known information about symbols in the expression
- Polymorphism
  - Single algorithm (function) can have many types
- Overloading
  - Symbol with multiple meanings, resolved when program is compiled

November 2021

CSE341 Lecture 4

slide 89