Programming Languages
CS 214

 ©Joel C. Adams. All Rights Reserved. Dept of Computer Science Calvin College

Review

Recall: A type consists of *data* and *operations*.

The set constructors:

- ________________ is the basis for *aggregates*;
- ________________ is the basis for *operations*; and
- ________________ is the basis for *sequences*.

These three provide a formal way to construct types:

→ Use the ______ and ____________ to represent the ____;

→ Use the ________ constructor to represent the ________ on the type.

 ©Joel C. Adams. All Rights Reserved.

Declarations

One purpose of types is to permit objects to be _________.

– A declaration states the _________ of an object (for the compiler).

– Example: If an object's value may vary, it is a _________; otherwise it is a _________.

- Issue: Where may declarations occur?
  - Many languages (Ada, C) restrict the location of declarations:

    <ada-program> ::= procedure identifier ; <declaration-section> <block> ;

    C++ and Lisp are unusual in allowing declarations "anywhere"...
  - In Smalltalk they must appear near the start of a method or block:

    <smalltalk-block> ::= [<parameters> '|' <identifiers> '|' <expressions>]
  - Lisp declarations are functions/expressions, and so are permitted anywhere an expression may occur.

©Joel C. Adams. All Rights Reserved.

Constant Declarations

- Issue: How are constants distinguished from variables?
  - Most imperative languages use a special keyword:

<ada-const-dec> ::= identifier : constant <type> := <expression> ;

```
PI : constant real := 3.14159;
Mass, Energy : real;
```

  - C++ is similar, but uses the keyword *const*, and in a prefix form.

```
const double PI = 3.14159;
double mass, energy;
```

  - Java is similar to C++, but uses the keyword *final*.

```
final double PI = 3.14159;
double mass, energy;
```

  - Lisp constants can be declared using the *defconst* function.

```
(defconst PI 3.14159)
(defvar mass 0.0)
```

 ©Joel C. Adams. All Rights Reserved.

Imperative Issue

- Fortran-family languages (e.g., Fortran, C/C++ and Java) declare variables using a form like this: ______________; while Algol-family languages (e.g., Ada, Pascal, Modula-2) use a form like this: ________________;

Which approach is preferable?

```
double mass, energy;
```

1. The compiler 'knows' the type when it get to the ids

2. Each id in the list can be separately initialized:

```
double mass = 1.0,
       energy = 0.0;
```

```
mass, energy : real;
```

1. The compiler doesn't 'know' the type as it processes the ids.

2. Each id must be initialized elsewhere (or else all ids are limited to the same value):

```
mass, energy : real := 0.0;
```

 ©Joel C. Adams. All Rights Reserved.

Fundamental Type Operations

<table>
<tr><th>Operation</th><th>C++</th><th>Ada</th><th>Modula-2</th><th>Smalltalk</th><th>Lisp</th></tr>
<tr><td>+, -, *</td><td>+,-,*</td><td>+,-,*</td><td>+,-,*</td><td>+,-,*</td><td>+,-,*</td></tr>
<tr><td>real div</td><td>/</td><td>/</td><td>/</td><td>/</td><td>/</td></tr>
<tr><td>int div</td><td>/</td><td>/</td><td>DIV</td><td>//</td><td>/</td></tr>
<tr><td>modulus</td><td>%</td><td>mod</td><td>MOD</td><td>\\</td><td>mod</td></tr>
<tr><td>remainder</td><td></td><td>rem</td><td></td><td>rem:</td><td>rem</td></tr>
<tr><td>exponentiation</td><td></td><td>**</td><td></td><td>raisedTo:</td><td>expt</td></tr>
<tr><td>logical-and</td><td></td><td>and</td><td></td><td>&</td><td></td></tr>
<tr><td>short-circuit-and</td><td>&&</td><td>and then</td><td>AND</td><td>and:</td><td>and</td></tr>
<tr><td>logical-or</td><td></td><td>or</td><td></td><td>|</td><td></td></tr>
<tr><td>short-circuit-or</td><td>||</td><td>or else</td><td>OR</td><td>or:</td><td>or</td></tr>
<tr><td>not</td><td>!</td><td>not</td><td>NOT</td><td>not</td><td>not</td></tr>
<tr><td>equality</td><td>==</td><td>=</td><td>=</td><td>=, ==</td><td>=, eq, equal</td></tr>
<tr><td>inequality</td><td>!=</td><td>/=</td><td>#, <></td><td>~=, ~~</td><td>/=</td></tr>
<tr><td>relationals</td><td colspan="5">uniformly <,>,<=,>=</td></tr>
</table>

©Joel C. Adams. All Rights Reserved.

Short-Circuit Operators

___________, ___________ operators evaluate ____ operands.

By contrast:

- A _____________ operator only evaluates its second operand if its first operand is *false*; and
- A _____________ operator only evaluates its second operand if its first operand is *true*.

Short-circuit behavior is more time-efficient, and can be exploited in certain situations:

```
while (ptr != NULL && ptr->value != searchVal)
  ptr = ptr->next;
```

- If && were a logical-and instead of short-circuit-and, this condition would seg-fault when *searchVal* is not in the list…

 ©Joel C. Adams. All Rights Reserved.

Modeling Real-World Values

Suppose we want to model the seven "_______________.

One approach:

```
const int RED=0, ORANGE=1, YELLOW=2, GREEN=3,
          BLUE=4, INDIGO=5, VIOLET=6;
int aColor = BLUE;
```

This approach requires the human to map colors to integers…

Instead:

```
enum Color { RED, ORANGE, YELLOW, GREEN,
             BLUE, INDIGO, VIOLET } ;
Color aColor = BLUE;
```

Most imperative languages support such *enumerations*…

Ada:

```
type Color = ( RED, ORANGE, YELLOW, GREEN,
               BLUE, INDIGO, VIOLET ) ;
aColor : Color := BLUE;
```

An enumeration is a type whose values are explicitly listed.

 ©Joel C. Adams. All Rights Reserved.

Enumerations: Compiler Side

An enumeration's values must be valid *identifiers*:

and the compiler treats a declaration:

```
enum NewType { id0, id1, id2, …, idN-1 };
```

as being (approximately) equivalent to:

```
const int id0 = 0, id1 = 1, id2 = 2, …, idN-1 = N-1;
```

Thus, after processing

```
enum Color { RED, ORANGE, YELLOW, GREEN,
             BLUE, INDIGO, VIOLET };
```

so far as the compiler is concerned:

```
RED == 0 && ORANGE == 1 && YELLOW == 2 && … && VIOLET == 6
```

 ©Joel C. Adams. All Rights Reserved.

Enumerations: User Side

Enumerations thus provide an automatic means of mapping:

____

whose chief benefit is better program readability:

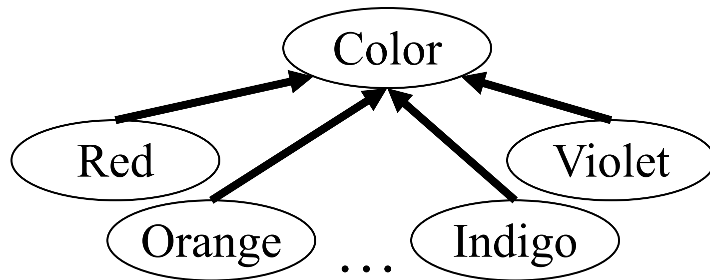
```
enum ElementName { HYDROGEN,
                   HELIUM,
                    …
                 };
ElementName anElement;
// ...
switch (anElement) {
  case HYDROGEN:
      atomicNumber = 1; break;
  case HELIUM:
      atomicNumber = 2; break;
  …
}
```

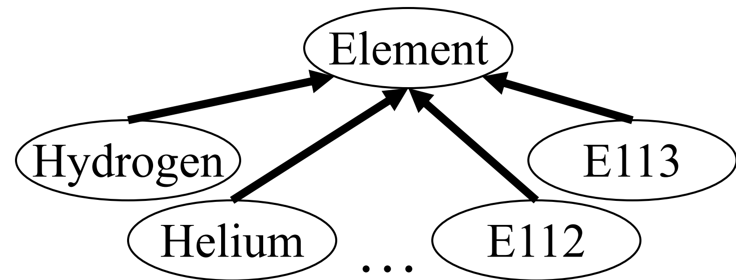
Enumerations allow real-world ‘values’ to be represented using real-world names, instead of (arbitrary) integer codes.

 ©Joel C. Adams. All Rights Reserved.

Enumerations and OO

OO purists replace enums with _________________:





This permits the creation of real-world _______:

```
// Smalltalk
aColor := new Blue.
```

```
// Smalltalk
anElement := new Helium.
```

as opposed to real-world _____ provided by an enumeration.

For this reason, “pure” OO languages like Smalltalk don’t provide an enumeration mechanism.

 ©Joel C. Adams. All Rights Reserved.

Subranges

Many imperative languages let us declare a __________: a type whose values are a subset of an existing type's values.

```
// Ada
subtype TestScore is Integer range 0..100;
subtype CapitalLetter is Character range 'A'..'Z';

type DaysOfWeek is (Sunday, Monday, Tuesday, Wednesday,
                    Thursday, Friday, Saturday);
subtype WeekDay  is DaysOfWeek range Monday..Friday;
```

If a subrange variable is declared:

```
WeekDay today;
```

and assigned an invalid value:

```
today := Saturday;
```

then an exception occurs that, if not caught, halts the system.

____________ is an essential feature for ____________________.

 ©Joel C. Adams. All Rights Reserved.

Sequence Types

There are two common structures for storing sequences:

- The _______ (and/or ______) that stores values in contiguous memory locations;
- The _________, that stores values anywhere there's room.

Arrays are _____________________: any value in the array can be accessed in the same amount of time…

– The address of the value at index i can be computed in _____:

  (arrayBaseAddress + (i - firstIndex) × ElementSize)

Once declared, the size of an ______ is usually ______.

A _______ is an array-like structure ________________.

 ©Joel C. Adams. All Rights Reserved.

Arrays

Most languages let us build arrays and/or vectors:

```
double anArray[8];                                  // C++
double * aVector = new double[n];
vector<double> anotherVector(n);
```

```
type ArrayType is array(1..8) of real;          -- Ada
anArray : ArrayType;
type Vector is array(integer range <>) of real;
type VectorAccess is access Vector;
aVector : VectorAccess;
begin
    aVector := new Vector(1..n);
```

```
(setq anArray (vector 0 1 2 3 4 5 6 7 8))  ”Clojure"
```

```
| anArray aVector |                             "Smalltalk"
anArray := Array new: 8.
aVector := OrderedCollection new: n.
```

 ©Joel C. Adams. All Rights Reserved.

Array Indexing

Arrays/vectors are ___________________________:
 the value stored at index *i* can be accessed in O(1) time:
 (arrayBaseAddress + (*i* - firstIndex) × ElementSize)

The default *firstIndex* is different in different languages:

- C/C++: _
- Ada: ________________________
- Lisp: __
- Smalltalk: __

At Issue: There is an ________________________________:

- Accesses to 0-relative arrays require one fewer operation:
  (arrayBaseAddress + (_____) × ElementSize)
  = arrayBaseAddress + _ × ElementSize
- Programmer-specified index values can be very convenient:

```
type LetterCounter is array(CapitalLetter) of integer;
type DailySales is array(WeekDay) of real;
```

 ©Joel C. Adams. All Rights Reserved.

Array Access

An important array operation is to access the value at index *i*.

There are two different flavors to this operation:

- the ______ version _______ the value at index *i*;
- the _____ version lets us ______ the value at index *i*.

Most languages use the same syntax for both operations:

- C/C++:

```
oldValue = anArray[i];                // read
anArray[i] = newValue;                // write
```

- Ada:

```
oldValue = anArray(i);                -- read
anArray(i) = newValue;                -- write
```

- Lisp:

```
(setq oldValue (aref anArray i))      "read"
(setq (aref anArray i) newValue)      "write"
```

Other languages provide distinct operations for the two:

- Smalltalk:

```
oldValue := anArray at: i.            "read"
anArray at: i put: newValue.          "write"
```

 ©Joel C. Adams. All Rights Reserved.

Array Access (ii)

Although C++ uses the same syntax for both read and write, the operations are implemented as ________________:

```
template<class Type>
class vector {
 public:
   …
   // returns a read-only reference to v[i]
   const Type& operator[](unsigned i) const; // read version

   // returns a writable reference to v[i]
   Type& operator[](unsigned i);             // write version
   …
};
```

The compiler links a given call to the proper function:

```
oldValue = anArray[i];                // linked to read version
anArray[i] = newValue;                // linked to write version
```

 ©Joel C. Adams. All Rights Reserved.

Aggregates: Records/Structs

Like an array, an _________ type can store multiple values; but (unlike an array) it can store values of different types.

```
struct Student {    // C++
  int id;
  bool fTime;
  double gpa;
};
```

```
type Student is    -- Ada
  record
    id : integer;
    fTime: boolean;
    gpa : float;
  end record;
```

_______ has _______________, as its classes can do anything a record/struct can do...

```
"Lisp"
(defstruct Student
   id fTime gpa)
```

Once we have an aggregate type, we can build variables:

```
// C++
Student stu;
```

```
-- Ada
stu : Student;
```

```
"Lisp"
(setq stu make-Student)
```

 ©Joel C. Adams. All Rights Reserved.

Record/Struct Projection

We can then ________________ of the record/struct:

```
// C++
stu.id = 1234;
stu.fTime = true;
stu.gpa = 3.0;
```

```
-- Ada
stu.id:= 1234;
stu.fTime:=true;
stu.gpa:= 3.0;
```

```
"Lisp"
(setq (Student-id stu) 1234)
(setq (Student-fTime stu) t)
(setq (Student-gpa stu) 3.0)
```

Most languages use similar syntax to retrieve a field's value:

```
// C++
cout
 << stu.id;
 << stu.fTime
 << stu.gpa;
```

```
-- Ada
put(stu.id);
put(stu.fTime);
put(stu.gpa);
```

```
"Lisp"
(princ (Student-id stu))
(princ (Student-fTime stu))
(princ (Student-gpa stu))
```

These represent the __________ operation in each language...

 ©Joel C. Adams. All Rights Reserved.

Pointers

Most languages permit a programmer to define variables that can store an ________: also known as ______________.

These can be used to build lists of linked nodes:

```
// C++
struct Node {
  SomeType value;
  Node * next;
};
```

```
-- Ada
type Node;
type NodePtr is access Node;
type Node is record
  value: SomeType;
  next: NodePtr;
end record;
```

Smalltalk and Java have no pointer types because

____

____

Clojure variables are also pointers:

```
"Clojure"
(defstruct Node
   value next)
```

 ©Joel C. Adams. All Rights Reserved.

Dynamic Allocation & Deallocation

Each language supports ______________________________,
most commonly via the *new* operation:

```
// C++
Node * nPtr = new Node;
```

```
-- Ada
nPtr: NodePtr := new Node;
```

```
"Smalltalk"
| nPtr |
nPtr := Node new.
```

```
"Lisp"
(setq nPtr (make-Node))
```

Many languages also provide a _____________ operation:

```
// C++
delete nPtr;
```

Ada, Lisp, and Smalltalk instead provide a ___________ that eliminates memory leaks by automatically reclaiming unused memory (i.e., when no pointers to it exist).

 ©Joel C. Adams. All Rights Reserved.

Pointer Assignment and Copying

Each language provides a means of ____________________:

```
// C++
Node * tempPtr;
tempPtr = nPtr;
```

```
-- Ada
tempPtr: NodePtr;
tempPtr := nPtr;
```

```
"Smalltalk"
| nPtr tempPtr|
…
tempPtr := nPtr.
```

```
"Clojure"
(setq tempPtr nPtr)
```

Most languages also allow ________________________:

```
// C++
tempPtr = new Node;
*tempPtr = *nPtr;
```

```
-- Ada
tempPtr := new Node;
tempPtr.all := nPtr.all;
```

```
"Smalltalk"
tempPtr := nPtr copy;
```

```
"Lisp"
(setq tempPtr (copy-Node nPtr))
```

 ©Joel C. Adams. All Rights Reserved.

Other Pointer Operations

Languages provide a ______________________________:

```
// C++
nPtr = nullptr;//NULL;
```

```
-- Ada
nPtr:= null;
```

```
"Smalltalk"
nPtr := nil.
```

```
"Lisp"
(setq nPtr nil)
```

Most languages also permit ___________________________:

```
// C++
nPtr->value = 1234;
nPtr->next = new Node;
```

```
-- Ada
nPtr.value:= 1234;
nPtr.next:= new Node;
```

Smalltalk "fields" are accessed via accessor and mutator methods

```
"Lisp"
(setq (Node-value nPtr) 1234)
(setq (Node-next nPtr) (make-Node))
```


©Joel C. Adams. All Rights Reserved.

Type Systems

A __________ is a set of rules by which a language associates types with expressions.

The system generates a _________ when its rules do not permit a type to be associated with an expression.

Example: Early Fortran versions had only integers and reals.

- Declarations not required: ________ of identifiers
  - Identifiers beginning with I-N are integers; all others are reals.
- Literals with decimal points are real; others are integers.
- Type System Rule: If E1 and E2 are expressions of the same type T, then E1+E2, E1-E2, E1*E2, and E1/E2 produce a result of type T.
  - I+N produces a value of type integer; X+Y produces a value of type real.
  - Expressions like X+I (e.g., 0.5+1)or N-Y generate __________.

 ©Joel C. Adams. All Rights Reserved.

Type System Formalism

If f is a function from $(S) \rightarrow T$, and $s \in S$, then $f(s) \in T$.

Ada defines:	$+(\text{int} \times \text{int}) \rightarrow \text{int}$
and	$+(\text{real} \times \text{real}) \rightarrow \text{real}$
but neither	$+(\text{real} \times \text{int}) \rightarrow \text{real}$ nor $+(\text{int} \times \text{real}) \rightarrow \text{real}$
so both	2 + 3 and 2.0 + 3.0 are valid expressions;
but neither	2.0 + 3 nor 2 + 3.0 are valid expressions.

Arithmetic expressions mixing reals and ints cause type errors.

Ada's other arithmetic operators behave the same way.

Why would Ada's designers choose such a type system?

- Ada is designed for building ___________________...
- Ada's type system is perhaps the most strict of any HLL.
- Ada compilers catch errors that slip by in other languages.

 ©Joel C. Adams. All Rights Reserved.

Coercion

Ada is unusual in rejecting mixed-type arithmetic expressions; its goal is to prevent the unwanted loss of information.

Most HLLs permit arithmetic types to be freely intermixed.

To prevent information loss,

such languages take an expression:	2 + 1.5
“expand” the “smaller” operand:	2.0 + 1.5
and then perform the “larger” operation:	$+(real \times real)$

The automatic conversion of an operand’s type to prevent rejection by the type system is called a ______________.

Some languages use the term __________ to describe this; others describe it as _________.

 ©Joel C. Adams. All Rights Reserved.

Overloading

Note: operators like +, -, *, ... are _______________. In $a + b$:

+ means "perform integer addition" if a and b are integers;

+ means "perform real addition" if a and b are reals.

___________ symbols have different meanings in different contexts.

Formally: For any function $f(D) \rightarrow R$:

– The set of all possible arguments D is the function's _________;

– The set of all possible results R is the function's _______.

An overloaded function is _________________________________.

+, -, *, / are overloaded in most HLLs

To process such operations, the compiler must check the context (operand types) and find a definition with the matching domain.

A _________ occurs when no definition with that domain is found.

 ©Joel C. Adams. All Rights Reserved.

Type Checking

A type system enforces its rules by _______________:

- Analyzing the code, looking for type errors
- Only permitting programs without type errors to execute.
- A program with no type errors is described as __________.

Type checking is accomplished at two levels:

1. _________________: check for type-errors at ____________.
2. _________________: check for type-errors at ____________.

Ada performs both static and dynamic checking, but the language is designed to maximize the number of errors that can be detected statically (i.e, by the compiler).

 ©Joel C. Adams. All Rights Reserved.

Static Checking Examples

a. In C++ expressions of the form: x % y

the compiler looks up the types of x and y
(in a data structure called the ____________)
and rejects the expression if both are not of type int.

b. In C++ expressions of the form: sqrt(x)

the symbol table contains both the type T of argument x
and the domain-set D for which sqrt() is defined,
allowing the compiler to reject the expression if $T \notin D$.

– Original (K&R) C did not require that function prototypes contain parameter types, making it impossible for the compiler to type-check function calls (ANSI-C corrected this).

 ©Joel C. Adams. All Rights Reserved.

Dynamic Checking Examples

Dynamic checking is checking for errors only detectable at run-time by inserting checks before an expression's code.

Expression: x / y

```
// without dynamic checks
      mov x, R0
      div R0, y
```

```
-- with dynamic checking
      mov x, R0
      mov y, R1
      cmp R1, #0
      be  DivideByZero
      div R0, R1
```

A[i]

```
// without dynamic checks
      mov A, R0
      add R0, i
```

```
-- with dynamic checking
      mov A, R0
      mov i, R1
      cmp R1, firstIndex
      blt  IndexTooLow
      cmp R1, lastIndex
      bgt IndexTooHigh
      add R0, R1
```

Dynamic checking is ________ and ______________...

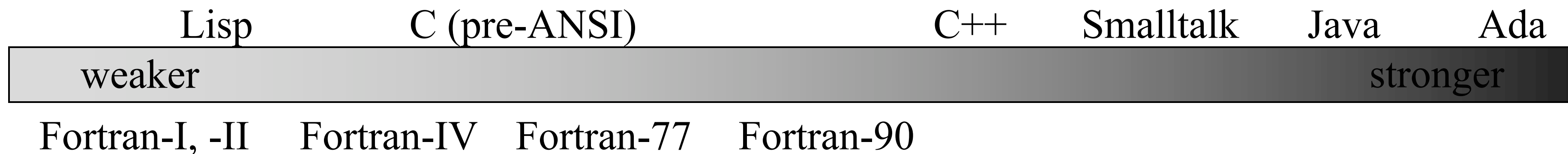
©Joel C. Adams. All Rights Reserved.

Type Strength

A language is ______________ if it has a strict type system.

A language is ______________ if it has a loose type system.

From this perspective, languages lie somewhere on a continuum, based on the the strength of their type system:



Language type systems have tended to get stronger as they evolve through different versions.

- The importance of type-strength has increased as the systems being built have increased in size, complexity, and importance.

 ©Joel C. Adams. All Rights Reserved.

Type Compatibility

What determines if two types *T1* and *T2* are compatible (e.g., can *T1* arguments be passed to *T2* parameters)?

```
typedef int IntArray[32];
IntArray x, y;
// Are x, y compatible?
```

```
int x[32];
void f(int y[32]);
// Are x, y compatible?
```

```
struct Student {                struct Employee {
  int id;                          int id;
  string name;                     string name;
};                              };
Student stu;                    Employee emp;
// Are stu, emp compatible?
```

```
struct Student {                struct Employee {
  int studentID;                   int empID;
  string studentName;              string empName;
};                              };
Student stu;                    Employee emp;
// Are stu, emp compatible?
```

 ©Joel C. Adams. All Rights Reserved.

Equivalence

Compatibility depends on whether a language views two types as ____________.

There are two broad categories of equivalence:

- Languages that use ____________ view two types as equivalent if they have *the same memory structure*.
- Languages that use ____________ view two types as equivalent if they are declared using the same *name*.

To illustrate, suppose that we have these declarations:

```
struct Student {                    struct Employee {
  int studentID;                       int empID;
  string studentName;                  string empName;
};                                  };
Student stu;                        Employee emp;
// Are stu, emp equivalent?
```

 ©Joel C. Adams. All Rights Reserved.

Structural Equivalence (SE)

Structural equivalence relies on three "rules":

- SE1: *A type name is structurally equivalent to itself.*

```
Student stu1;
Student stu2;
```

→ Since their types have the same name, *stu1* and *stu2* are structurally equivalent.

- SE2: *Two types formed by applying the same constructor to SE types are structurally equivalent.*

```
Student stu;
Employee emp;
```

→ Since both are members of (int × string), *stu* and *emp* are structurally equivalent.

- SE3: *If one type is an alias of another, the two types are structurally equivalent.*

```
typedef Student Transfer;
Student stu;
Transfer trans;
```

→ Since *Transfer* is an alias of *Student*, *stu* and *trans* are structurally equivalent.

©Joel C. Adams. All Rights Reserved.

Name Equivalence (NE)

There are different varieties of name equivalence:

- Pure NE: ______________________________.

```
-- Ada uses pure name equivalence
type IntArray is array(1..32) of Integer;
type IntList is array(1..32) of Integer;

a1: IntArray;
a2: IntList;
a3: IntArray;
a4: array(1..32) of Integer;
a5: array(1..32) of Integer;
```

→ Since __ and __ are declared with the same name, they are equivalent.

→ If we declare: `procedure print(IntArray anArray);` then Ada's type system will only accept __ and __ as arguments.

→ __'s type has a name, but it is a different name from the others.

→ __ and __'s types have no name: ______________ in Ada.

 ©Joel C. Adams. All Rights Reserved.

Name Equivalence (ii)

- Transitive NE: ______________________________.

```
-- C++ uses transitive name equivalence
struct Student {                         struct Employee {
  int idNumber;                             int idNumber;
  string name;                              string name;
};                                       };
typedef Student Transfer;
Student stu;                             Employee emp;
Transfer trans;
```

→ ___ and _____ are compatible; ___ is not compatible to either.

→ If we declare: `void print(Student aStudent);`

then the type system will only accept ___ or _____ as arguments, but will reject ____ as an argument.

 ©Joel C. Adams. All Rights Reserved.

Which is better?

- Consider type-checking on aggregates:
  - Type-checking is much simpler under name equivalence, as the type-checker just has to do a single comparison (*T1* == *T2*).
  - Under structural equivalence, the type-checker must do *an* ________________________________ (e.g., nested records).
- Name equivalence encourages abstraction:
  - NE encourages detail-hiding (ADT) by rejecting anonymous types:

```
procedure Put(seq: Sequence);
```

→ ____________________.

```
procedure Put(seq: array(1..32) of Integer);
```

→ ________.

  - SE discourages abstraction by accepting anonymous types:
    - SE may permit programs to be __________ (abstraction takes time).
    - Such programs may be _____________; may be ___________.

 ©Joel C. Adams. All Rights Reserved.

Summary

A type consists of ____ and ____________.

The set constructors: ________, __________, and __________ provide a formal way to represent type construction, using ________ and __________ to represent the ______, and __________ to represent the _____________ on the new type.

_______________ let us use __________________ for type data.

_______________ constrain the values of existing data types.

_______ and ________ are sequences stored in _______________ that permit _______ time access to any value.

_______ are sequences stored in ____________________, that require _________time (average) to access a random value.

©Joel C. Adams. All Rights Reserved.

Summary (ii)

____________ store ________________________________.

_________ store __________, permit us to build linked nodes.

A _________________ performs type-checking using _________________ or a version of _________________.

Type-checking may be:

- ______: done at _____________; and/or
- ______: done at _____________.

The more type-checking a language requires, the stronger its type-system, and the fewer type-errors slip past.

_____ has a very strong type-system.

 ©Joel C. Adams. All Rights Reserved.