4. Data Types
Type Systems, Type Checking, Records, Arrays, Strings, Pointers and Recursive Types, Files
Type Systems

- Definition mechanisms
- Associating types with variables and values
- Type equivalence
- Type compatibility
- Type inference
Not all languages have them (Fortran, C)

Some are limited
- C++/Java: every class defines a type – but this is the only method
  - Example:
    ```
    class Customer {...};
    ```

Some have a special keyword
- Ada: `type`
  - Example
    ```
    type Customer is record ... end record;
    type CustomerList is array [integer] of Customer;
    ```
Type Bindings

- **Variables**
  - **Java**: Customer c; Customer[] L;
  - **Ada**: c : Customer; L : CustomerList;

- **Values**
  - Every *value* must have a type, even if *variables* are *polymorphic*
  - Scheme/Lisp: *only* values have types; variables undeclared–can refer to values of *any* type
  - C++/Java: objects have a type; variables can refer only to objects of declared type or any subtype
Type Equivalence

- **Structural equivalence**
  - Two objects are of equivalent type if they are “built” the same way
  - Difficult to make precise
  - Examples of difficulty
    - `array[1..20] of integer` vs. `array[0..19] of integer`
    - `struct{int n, char c;}` vs. `struct{int c; char n;}`
    - `struct{char c; int n;}` vs. `struct{char c; int n;}

- **Name equivalence**
  - Two objects are of equivalent type if they refer to the same type name
  - What if type is anonymous?
Examples

- **Algol-descendant style**
  ```plaintext
type A is record integer n; end record;
type B is A;
type C is record integer n; end record;
aval1, aval2 : A; bval : B; cval : C;
anonval : record integer n; end record;
```

- **C style**
  ```plaintext
struct A {int n;};
struct C {int n;};
A aval1, aval2; C cval;
struct {int n;} anonval;
```

- **Structurally,** `aval1`, `aval2`, `cval`, and `anonval` (and `bval`) are all of equivalent type
- **Nominally,** `aval1` and `aval2` (and maybe `bval`) are of equivalent type
Type Compatibility

- Defines when values of one type can be used where another (non-equivalent) type is expected
  - Explicit type conversion (*casting*)
    Example (Java)
    ```java
    int x = 2.3; // illegal
    int x = (int)2.3; // OK
    ```
  - Implicit type conversion (*coercion*)
    Example (C/C++)
    ```c/c++
    int x = 2.3; // OK — 2.3 “coerced” to int
    ```
  - C++ even allows programmer-defined coercions
    Examples: copy constructor, `operator int`
Type Inference

- Determination of type in absence of a declaration
- Not the same as untyped
  - If type is inferred, the type of the variable is known and enforced. Must be unambiguously obtainable from program text
  - Untyped languages allow variable to refer to any type of value. Variable itself has no type
Expressions in C/C++/Java
- Given int x; double y; infer type of 2*x + 3*y as double
  - 2 and x are int, therefore so is 2*x
  - 3 is int, y is double, therefore 3*y is double
  - 2*x is int, 3*y is double, therefore 2*x + 3*y is double

Functions in Haskell
- Definitions
  f x = [x, x] -- f(x) = list of 2 items, both x
  fc x = [x,'a'] -- fc(x) = list of 2 items, x and 'a'
  f1 x = [x, 1] -- f1(x) = list of 2 items, x and 1
  g x = x + 2 -- g(x) = x + 2

Type inferences
f :: a -> [a] (Any kind of item can be listed)
fcs :: Char -> [Char] ('a' is in list, so only characters allowed)
f1s :: Num a => a -> [a] (1 is in list, so only numbers allowed)
g :: Num a => a -> a (Any numbers can be added)
If variables are typed, correct type generally checked statically (compile time)

Language that checks all types is strongly typed

“Runtime type checking” applies to languages (such as Python) which leave variables untyped

- Values have types
- Variables may refer to any value of any type
- Correctness of usage (e.g., in `a + b`, are `a` and `b` of type that can be added?) checked at runtime
Records

- Composite type containing *tuple* of values
  - Heterogeneous
    - Can contain elements ("fields") of different types
  - Contiguous
    - No unnecessary gaps within storage
    - Machine rules may force some gaps
  - Component selection using identifiers
    - Each field has a *name*
- Called *structures* in many languages
- Basis of *class* construct
  - A class is a record – however, it contains not only data but also functions (methods)
Issues

- Syntax of selection
  - Ada / C
    customer.name, customer.id, customer.name.name.last
  - Cobol
    name of customer, id of customer, last of name of customer or just last of customer or even last if unambiguous

- Union types
  - Example (C): `union {int i; char c}` means not two separately stored fields, but i and c stored in same location
  - Dangerous – should it be allowed? If allowed, should a tag be required?
    - Example (Pascal):
      ```pascal
      record
        case tag: boolean of
          true: (i integer); false: (c: char);
      end;
      ```
Composite type containing tuple of values

- Homogeneous
  - Elements must be all of same type

- Contiguous
  - No gaps within storage

- Component selection using computable indices
  - Since elements are of same type, they have same size
  - Since size is same for all and there are no gaps in storage, if array $a$ starts in location $addr$, the location of element $i$ of $a$ is $addr + i \times size$
Issues

- What constitutes type
  - Size may or may not be part of type
    - Pascal – yes
    - C/C++/Java – no
  - Important for functions

- Syntax of selection
  - Algol and most descendants – \texttt{a[i]}
  - Ada/Cobol/PL/I – \texttt{a(i)}
  - Haskell – \texttt{a!i}

- Multidimensional case
  - C/C++/Java – array of arrays: \texttt{a[i][j]}
  - Pascal – same, but can use \texttt{a[i][j]} or "pretend" 2-D: \texttt{a[i, j]}
  - Ada – either possible, meanings different (no pretending)
Issues (continued)

- Binding time of array size
  - C/C++
    - Automatic arrays must be declared with constant size
    - (Java doesn’t have automatic arrays)
  - Algol
    - Possible to wait until runtime
      
      ```c
      read(n);
      array a[1:n];
      ```

- Index range
  - C/C++/Java – start at 0
  - Algol, etc. – both upper and lower bounds specifiable
Generally presented as ADT

Implementations

- Array of characters – but then need to mark end of string
  - Example: C – end marked with ‘\0’
- Record including array of characters and length
  - Example: Java and C++ libraries

In language with encapsulation, strings accessible through methods only
  - Example: Java library
Pointers and Recursive Types

- Indirect addressing done using pointers
  - Needed for references within value model
- Recursive types possible if object has component which refers to another object of the same type
- In mixed models pointers are needed
- In pure reference model, no special notation is required
- Syntactic issue: if declaration before use is needed, some decision needed on method
Examples

- **Value model with pointers: C++**

```cpp
class Node {
    int data;
    Node * next;
};
Node n; // value
n.data = 3;
n.next->data = 5;
```

- **Reference model: Java**

```java
class Node {
    int data;
    Node next;
}
Node n; // reference
n.data = 3;
n.next.data = 5;
```
Encapsulation of an operating system entity

Difficult to make completely independent of implementation

Since Algol, generally provided as library, rather than built into language
  - C, C++, and Java

Some older languages have actual language syntax for files
  - FORTRAN, COBOL, Basic