# Imperative Programming: Data Types

## (Based on [Sethi 1996])

Yih-Kuen Tsay

Department of Information Management
National Taiwan University

# Data in Imperative Programming

- The emphasis is on data structures with assignable components.
- The size and layout of data structures tend to be fixed at compile time.
- Dynamic data structures are implemented using fixed-size cells and pointers.
- Allocation and deallocation of storage are explicit.

# Types

- An *object* is something meaningful to an application.
- *Data representation* refers to the organization of values in a program.
- Objects in an application have corresponding (data) representations in a program.
- Data representations in imperative languages are built from *values* that can be manipulated directly by the underlying machine.
- Values held in machine locations can be classified into *basic types*, such as integers, characters, reals and booleans.
- *Structured types* can be built up from simpler types and are laid out using sequences of locations in the machine.
- *Type expressions* (or simply *types*) are used to lay out values in the underlying machine and to check that operators are applied properly within expressions.

# First-Class Values

- Basic values (values of basic types) such as integers are first-class citizens. They can
  - be denoted by a name,
  - be the value of an expression,
  - appear on the right side of an assignment,
  - be passed as parameters, etc.
- Operations on basic values are built into the languages (and implemented efficiently).

# Types in Pascal

IM NTU

$$
\begin{array}{lcl}
\langle \mathit{simple} \rangle & ::= & \langle \mathit{name} \rangle \\
 & | & \langle \mathit{enumeration} \rangle \\
 & | & \langle \mathit{subrange} \rangle \\
\langle \mathit{type} \rangle & ::= & \langle \mathit{simple} \rangle \\
 & | & \textbf{array}\ [\ \langle \mathit{simple} \rangle\ ]\ \textbf{of}\ \langle \mathit{type} \rangle \\
 & | & \textbf{record}\ \langle \mathit{field\_list} \rangle\ \textbf{end} \\
 & | & \textbf{set of}\ \langle \mathit{simple} \rangle \\
 & | & \uparrow \langle \mathit{name} \rangle \\
\langle \mathit{enumeration} \rangle & ::= & (\ \langle \mathit{name\_list} \rangle\ ) \\
\langle \mathit{subrange} \rangle & ::= & \langle \mathit{constant} \rangle\ ..\ \langle \mathit{constant} \rangle \\
\langle \mathit{field} \rangle & ::= & \langle \mathit{name\_list} \rangle\ :\ \langle \mathit{type} \rangle
\end{array}
$$

# Basic Types

- Enumerations
- Integers and Reals
- Booleans and Boolean Expressions
- Subranges

| Operators of Pascal | Operators of C |
|---|---|
| $< \;\; <= \;\; = \;\; <> \;\; >= \;\; >$ in | $\|\|$ |
| $+ \;\; -$ or | && |
| $* \;\; /$ div mod and | $== \;\; !=$ |
| not | $> \;\; < \;\; <= \;\; >=$ |
| | $+ \;\; -$ |
| | $* \;\; / \;\; \%$ |
| | ! |

# Enumerations



- An *enumeration* is a finite sequence of names written between parentheses (in Pascal). The declaration

$$\textbf{type}\ day = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);$$

  makes *day* an enumeration with seven elements.
- Names like *Mon* are treated as constants.
- Pascal and C insist that a name appear in at most one enumeration.
- The basic types **boolean** and **char** in Pascal are treated as enumerations.
- The elements of an enumeration are ordered.
- Operations on enumerations (in Pascal): $ord(x)$, $succ(x)$, and $pred(x)$.

# Short-Circuit Evaluation

- C and Modula-2 (Pascal's successor) use short-circuit evaluation for boolean operators.
- In the following C program fragment

```
while ( i >= 0 && x != A[i] )  i = i-1;
```

  control reaches the text `x != A[i]` only if the expression `i >= 0` evaluates to true.

# Characters and Type Conversion

- In C, characters are implicitly converted, or coerced, to integers.

```
#include <stdio.h>
main() { /* copy input to output */
  int c;
  c = getchar();
  while (c != EOF) {
    putchar(c);
    c =getchar();
  }
}
```

- Conversion between characters and integers must be done explicitly in Pascal. Function $ord(c)$ maps a character $c$ to an integer $i$; the inverse operation $chr(i)$ maps the integer $i$ back to the character $c$.

$$c = chr(ord(c))$$
$$i = ord(chr(i))$$

# Arrays

- An array is a data structure that holds a sequence of elements of the same type.
- The fundamental property of arrays is that $A[i]$, the $i$th element of array $A$, can be accessed quickly, for any value $i$ at run time.
- An array type specifies the index of the first and last elements of the array and the type of all elements.
- Pascal allows the array index type to be an enumeration or a subrange.

  **array** [1996..2000] **of real**
  **array** [($Mon$, $Tue$, $Wed$, $Thu$, $Fri$)] **of integer**
  **array** [**char**] **of** *token*

- Do array types include array bounds?

# Array Layout

- The *layout* of an array determines the machine address of an element $A[i]$ relative to the address of the first element. Layout can occur separately from *allocation*.

  **var** $A$ : **array** $[low..high]$ **of** $T$

- Assume that each element of type $T$ occupies $w$ locations. If $A[low]$ begins at location $base$, then $A[low+1]$ begins at $base + w$, $A[low+2]$ begins at $base + 2 * w$, and so on.
- A formula for the address of $A[i]$ is best expressed as

$$i * w + (base - low * w)$$

  where $i * w$ has to be computed at run time, but where $(base - low * w)$ can be precomputed.

# Using Arrays

**type** $token = (plus, minus, \cdots, number, lparen, rparen, \cdots);$
**var** $tok$ : **array** [**char**] **of** $token$;
The array $tok$ is initialized by assignments like

$$tok['+'] := plus;$$
$$tok['-'] := minus;$$

A program segment:

**case** $ch$ **of**
$'+', '-', '*', '/', '(', ')', ';'$: **begin**
    $lookahead := tok[ch]$;
    $ch :=\ '\ '$
  **end**;
$'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'$: **begin**
    $\ldots$
    $lookahead := number$
  **end**
**end**

# Array of Arrays

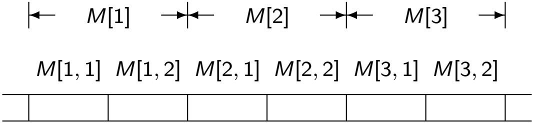
**var** $A$ : **array** $[low_1..high_1]$ **of array** $[low_2..high_2]$ **of** $T$
or **var** $A$ : **array** $[low_1..high_1, low_2..high_2]$ **of** $T$

## Row-major layout

The address of $A[i,j]$ is

$$i * rw + j * ew + (base - low_1 * rw - low_2 * ew),$$

where $rw$ is the width of a row $A[?, low_2..high_2]$ and $ew$ is the width of an element $A[?, ?]$.
Example: **var** $M$ : **array** $[1..3, 1..2]$ **of integer**



# Array of Arrays (cont.)

**var** $A$ : **array** $[low_1..high_1]$ **of array** $[low_2..high_2]$ **of** $T$
or **var** $A$ : **array** $[low_1..high_1, low_2..high_2]$ **of** $T$

## Column-major layout

The address of $A[i,j]$ is

$$i * ew + j * cw + (base - low_1 * ew - low_2 * cw),$$

where $cw$ is the width of a column $A[low_1..high_1, ?]$ and $ew$ is the width of an element $A[?, ?]$.

# Array Bounds and Storage Allocation

- Array layout (computation of array bounds) in C is done statically at compile time. Storage allocation is usually done upon procedure entry, unless the keyword `static` appears before a variable declaration.

```
int produce() {
static char buffer[128];
char temp[128];
    ...
}
```

- Storage for the static array `buffer` is allocated at compile time, while that for array `temp` is allocated afresh each time control enters procedure `produce`.

  Options for computing array bounds: *Static evaluation*, *Evaluation upon procedure entry*, and *Dynamic evaluation*.

# Records: Named Fields

- Records allow variables relevant to an object to be grouped together and treated as a unit.
- The type *complex* below is a record type with two fields, *re* and *im*:

$$
\begin{aligned}
\textbf{type}\ \mathit{complex} = \ & \textbf{record} \\
& \quad \mathit{re} : \textbf{real}; \\
& \quad \mathit{im} : \textbf{real}; \\
& \textbf{end};
\end{aligned}
$$

- The record type *complex* is simply a template for two fields *re* and *im*. Storage is allocated when the template is applied in a variable declaration, not when the template is described.
- A change in the order of the fields of a record should have no effect on the meaning of a program.
- Operations on records: selection and assignment.

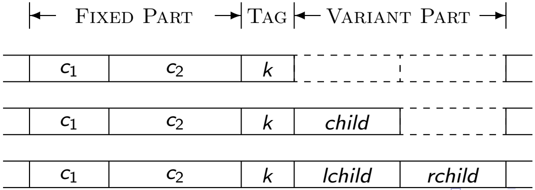
# Arrays vs. Records

| | *arrays* | *records* |
|---|---|---|
| component types | homogeneous | heterogeneous |
| component selectors | indices evaluated at run time | names known at compile time |

# Variant Records

- *Variant records* have a part common to all records of that type, and a variant part, specific to subsets of the records. Example:

$$
\begin{array}{l}
\textbf{type}\ \mathit{kind} = (\mathit{leaf}, \mathit{unary}, \mathit{binary});\\
\qquad \mathit{node} = \textbf{record}\\
\qquad\qquad\qquad c_1 : T_1;\\
\qquad\qquad\qquad c_2 : T_2;\\
\qquad\qquad\qquad \textbf{case}\ k : \mathit{kind}\ \textbf{of}\\
\qquad\qquad\qquad\quad \mathit{leaf} : (\ );\\
\qquad\qquad\qquad\quad \mathit{unary} : (\mathit{child} : T_3);\\
\qquad\qquad\qquad\quad \mathit{binary} : (\mathit{lchild}, \mathit{rchild} : T_4)\\
\qquad\qquad \textbf{end};
\end{array}
$$



# Variant Records and Type Safety

IM NTU

**type** *kind* = 1..2;
    *t* = **record**
        **case** *kind* **of**
            1 : (*i* : **integer**);
            2 : (*r* : **real**)
      **end**;
**var** *x* : *t*

An unsafe program segment:

    *x.r* := 1.0;
    *writeln*(*x.i*)

# Sets

- Pascal allows sets to be used as values. It also provides a type constructor **set of** for building set types from enumerations and subranges.
  - Set Values
    $[\,]$, $['0'..'9']$, $['a'..'z','A'..'Z']$, $[Mon..Sun]$, etc. All set elements must be of the same simple type.
  - Set Types
    The type "**set of** $S$" represents subsets of $S$.
  - Implementation
    A set of $n$ elements is implemented as a bit vector of length $n$.
  - Set Operations
    $x$ **in** $B$; $A + B$, $A - B$, $A * B$, $A/B$; $A \leq B$, $A = B$, $A \neq B$, $A \geq B$.

# Using Sets

**if** *ch* **in** ['+', '−', '∗', '/', '(', ')', ';'] **then begin**

    *lookahead* := *tok*[*ch*];

    *ch* := ' '

**end**

**else if** *ch* **in** ['0'..'9'] **then begin**

    . . .

    *lookahead* := *number*

**end**

# Using Sets (cont.)

Compared to

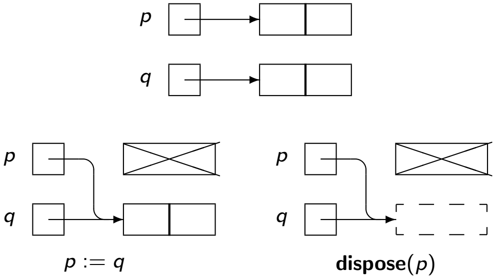
**case** *ch* **of**
$'+', '-', '*', '/', '(', ')', ';'$: **begin**
  $lookahead := tok[ch]$;
  $ch := \ ' \ '$;
 **end**;
$'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'$: **begin**
  ...
  $lookahead := number$
 **end**
**end**

# Pointers

- A *pointer* can have a value that provides indirect access to elements of a known type. Pointers are used for efficiency considerations and for manipulating dynamic data structures.
- Pointers are first-class values and can be used as freely as other values. They have a fixed size, independent of what they point to.
- Operations on pointers:
  - dynamic allocation on the heap: **new**($p$)
  - dereferencing: $p\uparrow$
  - assignment
  - equality testing
  - deallocation: **dispose**($p$)

# Dangling Pointers and Memory Leaks

- A *dangling pointer* is a pointer to storage that is being used for another purpose; typically, the storage has been deallocated.
- Storage that is allocated but is inaccessible is called *garbage*. Programs that create garbage are said to have *memory leaks*.
- Pointer assignment may result in memory leaks and **dispose** may result in dangling pointers.



# Types

- Type distinctions between values carry over to variables and to expressions.
  - Variable Bindings: A *variable binding* associates a property with a variable.
    - static binding (early binding)
    - dynamic binding (late binding)
  - Type Systems: A *type system* for a language is a set of rules for associating a type with expressions in the language.
    Rules of type checking:
    - function applications
    - overloading
    - coercion
    - polymorphism
  - Type Equivalence

# Type Equivalence

- A variable can be assigned the value of an expression or another variable of an equivalent type.
- When are two types equivalent?
  - In Pascal/Modula-2
    Type equivalence was left ambiguous in Pascal. Modula-2 defines two types to be *compatible* if
    1. they are the same name, or
    2. they are $s$ and $t$, and $s = t$ is a type declaration, or
    3. one is a subrange of the other, or
    4. both are subranges of the same basic type.
  - In C
    C uses structural equivalence for all types except records, which are called structures in C. Structure types are named in C and the name is treated as a type, equivalent to itself.

# Structural Equivalence

- The *structural equivalence* of two types is determined according to the following rules:
  1. A type name is structurally equivalent to itself.
  2. Two types are structurally equivalent if they are formed by applying the same type constructor to structurally equivalent types.
  3. After the type declaration **type** $n = T$, the type name $n$ is structurally equivalent to $T$.

# Type Checking

- The purpose of type checking is to prevent errors. A *type error* occurs if a function $f$ expects an argument of type $T$, but $f$ is applied to some $a$ that is not of type $T$.
- Questions to ask about type checking in a language:
  - Static or Dynamic
  - Strong or Weak

# Weak vs. Strong Type Checking