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.

DATA REPRESENTATION

<table>
<thead>
<tr>
<th>application</th>
<th>representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>data........</td>
<td>January 31</td>
</tr>
<tr>
<td>data........</td>
<td>May 6</td>
</tr>
<tr>
<td>variable....</td>
<td>d</td>
</tr>
<tr>
<td>operation..</td>
<td>.tomorrow(d)</td>
</tr>
</tbody>
</table>

Is this a good representation for date?
- Ambiguity between leap year and non-leap years!
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

\[
\begin{align*}
\langle \text{simple} \rangle & ::= \langle \text{name} \rangle \\
& \quad | \langle \text{enumeration} \rangle \\
& \quad | \langle \text{subrange} \rangle \\
\langle \text{type} \rangle & ::= \langle \text{simple} \rangle \\
& \quad | \langle \text{array} [ \langle \text{simple} \rangle ] \rangle \text{ of } \langle \text{type} \rangle \\
& \quad | \langle \text{record} \langle \text{field_list} \rangle \rangle \text{ end} \\
& \quad | \langle \text{set of} \langle \text{simple} \rangle \rangle \\
& \quad | \uparrow \langle \text{name} \rangle \\
\langle \text{enumeration} \rangle & ::= ( \langle \text{name_list} \rangle ) \\
\langle \text{subrange} \rangle & ::= \langle \text{constant} \rangle .. \langle \text{constant} \rangle \\
\langle \text{field} \rangle & ::= \langle \text{name_list} \rangle : \langle \text{type} \rangle
\end{align*}
\]
Layout of arrays, records, and pointers

A: array [0.. 2] of T

R: record
  a: Ta; Tb; Tc;
end

p: ↑ T

A: set of [1 .. 5]

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

```pascal
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)`
- `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

```c
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.

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

CHARACTERS AND TYPE CONVERSION

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`.

\[
\begin{align*}
    c &= \text{chr}(\text{ord}(c)) \\
    i &= \text{ord}(\text{chr}(i))
\end{align*}
\]
FRAGMENTS OF STDIO.H

```c
... #ifndef EOF
#define EOF (-1)
#endif
...
#define stdin (&_iob[0])
#define stdout (&_iob[1])
...
extern int fclose(FILE *);
extern FILE *fopen(const char *, const char *);
...
```

FRAGMENTS OF STDIO.H (CONT.)

```c
extern int fprintf (FILE *, const char *, …)
extern int fscanf (FILE *, const char *, …)
extern int printf (const char *, …)
extern int scanf (const char *, …)
...
extern int getc (FILE *);
extern int getchar (void);
extern char *gets (char *);
extern int putc (int, FILE *);
extern int putchar (int);
extern int puts (const char *);
```
The coercion of characters into integers in C

```c
#include <stdio.h>
main() {
    int c;
    c = getchar();
    while (c != EOF) {
        // Coercion
    }
}
```

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 [(Mon, Tue, Wed, Thu, Fri)] of integer
- array [char] of token

Do array types include array bounds?
**USING ARRAYS**

type *token* = (*plus*, *minus*, *times*, *divide*, *number*, *lparen*, *rparen*, *semi*);
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;
    ... *lookahead* := *number*
  end
end
```

**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.
ARRAYS OF ARRAYS

var A : array [low_1 .. high_1] of array [low_2 .. high_2] of T

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

- Row-major layout

The address of A[i, j] is

\[ \text{base} + (i - \text{low}_1) \times \text{rw} + (j - \text{low}_2) \times \text{ew}, \]

where \( \text{rw} \) is the width of a row, and \( \text{ew} \) is the width of an element.

Example: var M : array [1..3, 1..2] of integer

\[
\begin{array}{ccc}
\end{array}
\]

- Column-major layout

The address of A[i, j] is

\[ \text{base} + (j - \text{low}_2) \times \text{cw} + (i - \text{low}_1) \times \text{ew} \]

where \( \text{cw} \) is the width of a column, and \( \text{ew} \) is the width of an element.
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.

```c
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`:

```c
type complex = record
    re : real;
    im : real;
end;
```

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

<table>
<thead>
<tr>
<th></th>
<th>arrays</th>
<th>records</th>
</tr>
</thead>
<tbody>
<tr>
<td>component types</td>
<td>homogeneous</td>
<td>heterogeneous</td>
</tr>
<tr>
<td>component selections</td>
<td>indices evaluated at run time</td>
<td>names known at compile time</td>
</tr>
</tbody>
</table>

VARIANT RECORDS

Variant records have a part common to all records of that type, and a variant part, specific to some subset of the records.

Example:

```
type kind = (leaf, unary, binary);
node = record
c1 : T1;
c2 : T2;
case k : kind of
    leaf : ( );
    unary : (child : T3)
    binary : (lchild, rchild : T4)
end;
```
A LAYOUT FOR A VARIANT RECORD

Fixed Part \rightarrow Tag Field \rightarrow Variant Part \rightarrow

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>child</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lchild</td>
</tr>
</tbody>
</table>

VARIANT RECORDS AND TYPE SAFETY

```pascal
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)
```

Spring, 2001 Types 25

Spring, 2001 Types 26

1.
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 ≤ B, A=B, A ≠ B, A ≥ B.

USING SETS

if ch in ['+', '-', '*', '/', '(', ')', ';'] then begin
  lookahead := tok[ch]
  ch := ''
end
else if ch in ['0'..'9'] then begin
  ···
  lookahead := number
end

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
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`
- assignment
- equality testing
- deallocation: `dispose(p)`

---

A Cell In A Linked-list Data Structure

```plaintext
type link = ^cell;
cell = record
    info : integer;
    next : link;
end
```

![Diagram of a cell in a linked-list data structure]

```
info ----> link to next cell
```
**Inserting a cell at the front of a linked list**

- `new(p)`; leaves `p` pointing to a newly allocated cell
- `p↑.info := i;` set info field of the cell
- `p↑.next := front;` set next field
- `front := p` move front, as in follow

**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
An excerpt from a Pascal program

Program index(input, output);

var e, f : entry

type termrep = record
  spell : array [0..99] of char ;
  length : integer;
end;
entryrep = record
  term : termrep;
  page : integer;
end;
entry = ↑entryrep;

THE EFFECT OF SWAPPING POINTERS

temp := e;
e := f;
f := temp

(a)Before

(b)After
## OPERATIONS ON ENTRIES

```pascal
while x ≠ 0 do begin
  writeln (x);
  repeat read(next) until next ≠ x;
  x := next;
end;
```

**Removing adjacent duplicates from a list.**

<table>
<thead>
<tr>
<th>procedure</th>
<th>readentry(e)</th>
<th>Read an entry e.</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>endmarker(e)</td>
<td>Is e the end marker?</td>
</tr>
<tr>
<td>procedure</td>
<td>writeentry(e)</td>
<td>Write an entry e.</td>
</tr>
<tr>
<td>function</td>
<td>equalentry(e,f)</td>
<td>Are e and f equal?</td>
</tr>
<tr>
<td>procedure</td>
<td>copyentry(e,f)</td>
<td>Update e from f.</td>
</tr>
</tbody>
</table>

```pascal
readentry(e)
while no endmarker(e) do begin
  writeentry(e);
  repeat readentry(f) until not equalentry(e,f)
  copyentry(e,f);
end;
```

## A String Table layout from a Pascal program

<table>
<thead>
<tr>
<th>start</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>pool</td>
<td>T</td>
<td>e</td>
<td>x</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
**A String Table layout in a C program**

```
start

pool

[0 1 2 3 4 5]

EOS C | P | L | EOS B | C | P | L | EOS C | EOS C | + | + EOS

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

```c
p=start[next];
q=buffer;
for ( ; ; ) {
    *p = *q;
    if (*p == EOS) break;
    p++;
    q++;
}
```

Copy a string to a buffer.

---

**ARRAY AND POINTERS IN C**

```
p = &x;
x = x + 1;
*p = *p + 1;
```

For all arrays `a`, if `p` points to `a[i]`, then `p+1` is a pointer to `a[i+1]`.

In fact, an array name `a` is simply a pointer to the element `a[0]`.

```
a[0] = x;
i = n;
while (a[i] != x) {  
    -- i;
return i;
}
```

```
a[0] = x;
p = a+n;
while (*p != x) {  
    -- p;
return p - a;
}
```

```
int a[10];
int *Pa;
Pa = a;  √
a = Pa;  ×
Pa++;  √
a++;  ×
```

```
int a[10][20];
int *b[10];
```
**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)

– Imperative languages typically have static binding of types and dynamic bindings of values to variables.

• 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**

A variable can be assigned the value of an expression or another variable of an equivalent type.

```plaintext
expr1 := expr2
```

expr1 and expr2 must have the same type.

```plaintext
type T = array [1..100] of integer;
var x,y : array [1..100] of integer;
    z : array [1..100] of integer;
    w : T;
S: array [1..10, 1..10] of integer;
Do x, y, z, w, S have the same type?
```
TYPE EQUIVALENCE

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 $T$.

$x, y: array [0..9]$ of integer  
z : array [0..9]$ of integer
**PROBLEMS OF STRUCTURAL EQUIVALENCE**

```pascal
type dice = 1..12;
dozen = 1..12;
var x : dice;
y : dozen;
x and y are structurally equivalent, but conceptually we might want them to be different.
type complex = record re, im : integer end;
point = record x, y : integer end;

Determining structurally equivalence can get very complex for a Compiler.

record
A : dice;
B : ARRAY[1..5] of point;
end;

record
C : dozen;
D : ARRAY[1..5] of complex;
end;
```

---

**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

---

Spring, 2001 Types 43

Spring, 2001 Types 44