What is a data object?

- It is convenient to define the term *data object* as the runtime grouping of data within the virtual machine.
- Data objects can be programmer-defined (such as variables, constants, arrays, files, etc.) or system-defined (runtime-storage, stacks, subprogram activation records, file buffers, free-space lists, etc.).
Evolution of Data Type

• In early programming languages, data types were modeled exclusively after those native the computer, i.e, integers, reals and arrays.
• COBOL introduced decimal data values, followed by PL/I introducing length specifications and pointers.
• Since then records, unions and more have been introduced in newer languages with older languages incorporating new features as they continue to evolve.

What Is A Data Type?

• A data type is a set of values, together with a set of operations on those values having certain properties.
• Virtually every data object has expressible in a programming language has an implicit type:
  - `CONST Pi = 3.14159;` {Pascal}
  - `val x = 2;` ML
  - "hello, world" /* C */
Elementary and Aggregate Data Types

- A data object is elementary if it contains a data value that is always manipulated as a unit.
- A data object is a data structure if it is an aggregate of other data objects.

Data Objects And Their Bindings

- A data object will participate in several bindings in its lifetime.
- Although the attributes of a data object may remain fixed, the bindings may change dynamically.
- These bindings include:
  - **Type** – Associates the data object with the allowable set of values.
  - **Location** – may be changed by the memory management routine of the operating system.
  - **Value** – usually the result of an assignment operation.
  - **Name** – can be changed as a result of subprogram calls and returns
  - **Component** – data objects may be composed of several data objects. This binding may be represented by a pointer and subsequently changed.
Variables and Descriptors

• Variables are the implementation of data objects in programming languages
• A descriptor is the collection of a variable.
• In an implementation, a descriptor is the collection of memory cells that store the attributes of a variable.

Explicit Data Types and Sets of Values

• When we write in a C program
  
  ```c
  int x;
  ```

  we are explicitly declaring it to be of integer type.
• The values in the set are machine-dependent in C (defined in `limit.h`) but are explicitly set in Java as $-2^{31}$ to $+2^{31}-1$.
• The set of values can be explicitly listed (as for Java), enumerated, given as a subrange (as in Ada and Pascal) or borrowed from mathematics.
Data Types and Their Operations

- Operations include a few that are often overlooked, e.g.,
  \[ x[] \quad s\text{.salary} \quad s \rightarrow \text{name} \quad *s \]
- They may be expected to have specific properties, e.g.,
  \[ (x+1)-1 = x \quad \text{or} \quad x+y = y+x. \]

Simple Types

- Every language comes with a set of predefined data types; most allow new types to be constructed from the predefined types.
- Most predefined type are simple types, with no structure beyond their arithmetic and sequential structure.
- Some variations on simple types are predefined such as short, long, unsigned, long double.
Integers – Range Specification

- The set of integers values is a subset of the infinite integer set, with the bounds sometimes represented by a constant (such as Pascal’s `MaxInt`).
  - The range would be $-\text{maxint} - 1$ to $\text{maxint}$.
  - In C, there are four such specifications: `int`, `short`, `long`, and `char`.

Integers – Arithmetic Operator Specifications

The operations on integer data objects include:
- arithmetic – addition, subtraction, multiplication division and remainder, plus and minus.
  - BinOp: integer × integer → integer
  - UnaryOp: integer → integer
- assignment
  - Assignment: integer × integer → integer
    and
  - integer × integer → void
Integers – Logical Operator Specifications

The operations on integer data objects include:
• relational – equal to, not equal to, greater than, less than, greater than or equal, less than or equal
  – Relop : integer × integer → boolean
• Bit operations – and (\&), or (\|) and shifts (<< and >>)
  – integer × integer → integer

Integers- Implementation

• Integers are most commonly implemented using hardware-defined representations and operations.

<table>
<thead>
<tr>
<th>No descriptor</th>
<th>Sign bit</th>
<th>Binary integer</th>
<th>used in static type checking cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0110111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Descriptor stored in separate word</th>
<th>Type descriptor</th>
<th>Binary integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0110111</td>
</tr>
</tbody>
</table>

Used by LISP
Floating-Point Real Numbers Specifications

- A floating-point number may be specified by the attribute real (in FORTRAN or Pascal) or float (in C, C++ or Java) or by the exact precision that is required (in COBOL, PL/I or Ada).
- All the same arithmetic operations are available as in integers (except for remainder).
- Relational operations are somewhat restricted due to roundoff concerns.

Floating-Point Real Number Implementation

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>23 22</td>
</tr>
</tbody>
</table>

S = mantissa’s sign bit 0 is positive; 1 is negative
E = Exponent in excess-127 notation, i. e., E ranges from 0 to 255 but exponent of 2 ranges from –127 to +128
M = Mantissa of 23. Since mantissa is normalized, the most significant is assumed to be 1 and that gives a 24th significant bit.
Floating Point Numbers - Examples

+1 = \( 2^0 \times 1 = 2^{127-127} \times (1).0\) (binary) = 0 01111111 0000....
+1.5 = \( 2^0 \times 1.5 = 2^{127-127} \times (1).1\) (binary) = 0 01111111 1000...
-5 = \(-2^2 \times 1.25 = 2^{129-127} \times (1).01\) (binary) = 1 10000001 01000..

This gives real numbers the range \(10^{-38}\) to \(10^{38}\).

In 64-format, the exponent is 11 bits ranging from –1022 to +1023. Our numbers range from \(10^{-308}\) to \(10^{308}\)

Integers – Range Specification

- The set of integers values is a subset of the infinite integer set, with the bounds sometimes represented by a constant (such as Pascal’s \textit{MaxInt}).
  - The range would be \(-\text{maxint} - 1\) to \text{maxint}.
  - In C, there are four such specifications: \texttt{int}, \texttt{short}, \texttt{long}, and \texttt{char}. 
Integers – Arithmetic Operator Specifications

The operations on integer data objects include:
• arithmetic – addition, subtraction, multiplication division and remainder, plus and minus.
  – BinOp: integer× integer → integer
  – UnaryOp: integer → integer
• assignment
  – Assignment: integer× integer → integer
  and
  – integer× integer → void

Integers – Logical Operator Specifications

The operations on integer data objects include:
• relational – equal to , not equal to , greater than, less than, greater than or equal, less than or equal
  – Relop : integer× integer → boolean
• Bit operations – and (&) or (l) and shifts (<< and >>)
  – integer× integer → integer
Integers- Implementation

- Integers are most commonly implemented using hardware-defined representations and operations.

![Diagram of integer representation](image)

- No descriptor
- Descriptor stored in separate word

---

Floating-Point Real Numbers Specifications

- A floating-point number may be specified by the attribute real (in FORTRAN or Pascal) or float (in C, C++ or Java) or by the exact precision that is required (in COBOL, PL/I or Ada).
- All the same arithmetic operations are available as in integers (except for remainder).
- Relational operations are somewhat restricted due to roundoff concerns.
Floating-Point Real Number Implementation

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>23 22 0</td>
</tr>
</tbody>
</table>

S = mantissa’s sign bit 0 is positive; 1 is negative
E = Exponent in excess-127 notation, i.e., E ranges from 0 to 255 but exponent of 2 ranges from −127 to +128
M = Mantissa of 23. Since mantissa is normalized, the most significant is assumed to be 1 and that gives a 24th significant bit.

Floating Point Numbers - Examples

\[ +1 = 2^0 \times 1 = 2^{127-127} \times 1.0(binary) = 0 \ 01111111 \ 0000... \]
\[ +1.5 = 2^0 \times 1.5 = 2^{127-127} \times 1.1(binary) = 0 \ 01111111 \ 1000... \]
\[ -5 = -2^2 \times 1.25 = 2^{129-127} \times 1.01(binary) = 1 \ 10000000 \ 01000.. \]

This gives real numbers the range 10^{-38} to 10^{38}.

In 64-format, the exponent is 11 bits ranging from −1022 to +1023. Our numbers range from 10^{-308} to 10^{308}.
Decimal Numbers

- Most larger computers that are designed for business systems include hardware support for decimal numbers.
- Decimals hold a fixed number of decimal digits in fixed position.
- This were essential in COBOL and also exist in PL/I and C#.
- They are also known as Fixed Numbers.

Fixed-Point Real Numbers Specification

- There are certain problems where real numbers with a specific number of places are needed. Eg., monetary amounts.
- Fixed length numbers are represented as a number sequence of fixed length with a decimal point positioned.
- In COBOL, it is written as \texttt{X PICTURE 999V99}.
Fixed-Point Real Numbers
Implementation

• Fixed point may be supported by hardware or implemented by software emulation.
• Example, in PL/I
  
  \[
  \text{DECLARE } X \text{ FIXED DECIMAL}(10, 3),
  \]
  
  \[
  (Y, Z) \text{ FIXED DECIMAL}(10, 2);
  \]
  
  if we wrote \( x = 103.421 \), we would store 103, 421 together with a scale factor of 1000.
• Any and all arithmetic would have to keep track of scale factors.

Booleans

• Boolean or logical data objects represent the values true and false and implement the basic logical operations of comparisons, \textbf{AND}, \textbf{OR} and \textbf{NOT}.
• In Pascal and Ada, Boolean values can be viewed as an enumeration:
  
  \[
  \text{TYPE Boolean } = (\text{False, True});
  \]
• Their operations have the signatures:
  
  \[
  \text{and : Boolean } \times \text{ Boolean } \rightarrow \text{Boolean } \quad \text{(Conjunction)}
  \]
  
  \[
  \text{or : Boolean } \times \text{ Boolean } \rightarrow \text{Boolean } \quad \text{(Disjunction)}
  \]
  
  \[
  \text{not : Boolean } \rightarrow \text{Boolean } \quad \text{(Negation or complement)}
  \]
Implementing Booleans

- Booleans can be represented using
  - one bit to represent true (= 1) or false (= 0)
    with the rest of the byte ignored or
  - false being 0 and true being a nonzero value.
- Some languages like PL/I may make provisions
  for bit strings.
- C uses integer types but implements both bitwise
  (& and |) and logical operations (&& and ||)

Characters

- Character data is stored in one of several numeric
codes.
- The most common code is 8-bit ASCII (American
  Standard Code for Information Interchange),
  where 0-127 are used for printable and control
  characters and 128-255 for graphical characters.
- ISO 8859-1 is a different 8-bit code and is used by
  Ada 95.
- Unicode is a 16-bit code that includes characters
  from most the world’s natural languages. It is
  used by Java, JavaScript and C#.
Strings

• A string type is one where all the values are sequences of characters.

• There are two principle design issues:
  – Should strings be implemented as a type of character array (with subscripting operations) or as a primitive type?
  – Should strings have a static or dynamic length?

String Operations

• String operations typically include:
  – substring reference
  – catenation
  – relational operators
  – assignment
Strings in Different Languages

- C and Pascal treat strings as a special type of array.
  - In Turbo Pascal, a string is an array of up to 255 characters, where the zeroth element in the array holds its length.
  - In C, strings are arrays terminated by a null byte; references to string literals refer to its location in memory.
- Although C++ uses the C approach to strings, there is also a standard library class called string. Java also has a predefined class.
- FORTRAN 77, FORTRAN 90 and BASIC treat strings as a primitive type.

String Operations in Ada

```ada
r, s, t : string(1..30);
-- a predefined array type
begin
  r := "Hello, world";
  s := r(8:12); -- s = "world"
  t := r & s; -- t
       = "Hello, worldworld"
...
```
Why Enumerations?

- Sometimes we want a variable to hold one of a small numbers of symbolic values.
  - E.g, StudentClass can be Freshman, Sophomore, Junior, Senior.
- Older programming languages forced us to use integer values for this (e., Freshman, 1, Sophomore = 2, etc.)
- However, this doesn’t prevent us from using these for arithmetic operations for which they are not intended.
- Pascal, C and Ada all provide a mechanism for defining and manipulating these directly.

Enumerations - Specification

- An enumeration is an order list of distinct values, where the programmer defines both the names and their ordering:
  - In C
    ```
    enum studclass {freshman, soph, junior, senior};
    ```
  - In Pascal
    ```
    TYPE Class = (Freshman, Soph, Junior, Senior);
    VAR StudentClass : Class;
    ```
Using Enumerations

• Instead of writing
  \texttt{IF StudentClass = 3 THEN ... ...}
  you can write
  \texttt{IF StudentClass = \textit{Junior} THEN ... ...}
• In C, enumerations are assigned integer values and can be assigned to integer variables and used interchangeably.
• In Pascal, enumerations are each their own type and cannot be used interchangeably with each other or with native data types.

Implementing Enumerations

• Enumerations effectively are unsigned integers.
  – A 4-item enumerated type can be represented using 2 bits.
• Successor and Predecessor operations can be implemented using addition of 1 and subtraction by 1.
• C simply assigns integer values in sequence for these which can overridden by the programmer:
  – \texttt{enum class \{freshman = 14, soph = 36, jr = 72, sr = 96\};}
Enumerated Types in Ada

with Text_IO; use Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

PROCEDURE Enum is
  TYPE Color_Type is (Red, Green, Blue);
  -- define Color_IO so that Color_Type values
  -- can be printed
  package Color_IO is new Enumeration_IO(Color_Type);
    use Color_IO;
  x : Color_IO := Green;
begin
  x := Color_Type'Succ(x); -- x is now Blue;
  x := Color_Type'Pred(x); -- x is now Green;
  put(x); -- prints GREEN
  new_line
end Enum

Enumerated Types in C

#include <stdio.h>
enum Color (Red, Green, Blue)
enum NewColor    (NewRed = 3, NewGreen = 2,
                NewBlue = 2);
main()
{
  enum Color x = Green; /* x is actually 1 */
  enum NewColor y = NewBlue; /* y is actually 2 */
  x++; /* x is now 2, or Blue */
  --y; /* y is now 1 - not even in the enum*/
  printf("%d\n", x); /* prints 2 */
  printf("%d\n", y); /* prints 1 */
  return(0);
}
Enumerated Types in Pascal

PROGRAM UseEnum;
  TYPE
    Color = (Red, Green, Blue);
  VAR
    x, y : Color;
BEGIN
  x := Green; { Initialize x as Green }
  y := x; { Initialize y as Green }
  x := Pred(x); { x is now Red }
  y := Succ(y); { y is now Blue }
  WriteLn(Ord(x)); { Prints 0 }
  WriteLn(Ord(y)); { Prints 2 }
END. { UseEnum }

Subranges

• Subranges are contiguous subsets of simple types specified by the smallest and largest value in the range.
  type Digit_Type is range 0..9; -- Ada
  TYPE Digit = 0..9; { Pascal }
• Java doesn’t actually have subranges:
  byte digit; // digit can contain -128..127
  if (digit > 9 || digit < 0)
    throw new DigitException
• Most languages that support subranges require that they be ordinal, with a prespecified predecessor and successor.
• While Ada allows
  type Unit_Interval is range 0.0..1.0;
  Most other languages will not.
Subranges - Specification

• A subrange of type integer uses a subset of the range of integer values.
• It is usually declared
  A : 1..10 \textit{in Pascal}
  A : \texttt{integer range 1..10} \textit{in Ada}

Subrange Implementation

There are two important advantages to subranges:
• Smaller Storage Requirements
  – Many subranges can be represented using either a single bytes or a few bits.
  – C allows the use of characters to be manipulated as one byte unsigned integers.
• Better Type Checking
  – E.g., a month must have a values between 1 and 12.
Arrays

- In C, C++, and Java the index must be a nonnegative integer; this is not necessarily the case in Pascal (any ordinal type will do) or BASIC (real indices are truncated).

Arrays and Functional Languages

- Most functional languages do not have array data types, although a few do (these generally include imperative constructs).
- Scheme has a vector type and some versions of ML have an array module.
Declaring Array Types in C

- We can define types as:
  ```c
  typedef int TenIntArray[10];
  typedef int IntArray[];
  ```
- We can define variables as:
  ```c
  TenIntArray x;
  int y[5];
  int z[] = { 1, 2, 3, 4};
  IntArray w = [1, 2];
  ```
- The following are not legal:
  ```c
  const int Size = 5;
  IntArray w;
  int x[Size], y[Size*Size];
  ```

Arrays in C

```c
int f(int size)
{
    int a[size]; /* illegal */
    ...
}

/* indefinite size is legal*/
int array_max(int a[], int size)
{
    int temp, i;
    assert(size > 0);
    temp = a[0];
    for (i = 1; i < size; i++)
        if (a[i] > temp) temp = a[i];
    return(temp);
}
```
import java.io.*

// Arrays in Java are dynamically allocated using
// the heap and the size is specified. Changing it
// requires reallocating the array.

class arrayTest {
    static int array_max(int[] a) {
        int temp;
        temp = a[0];
        // size is part of a
        for (int i = 1; i < a.length; i++) {
            if (a[i] > temp) temp = a[i];
        }
        return temp;
    }
}

public static void main(String args[]) {
    // This placement of [] is also allowed
    System.out.print("Input a positive integer: ");
    // Java code to get formatted input
    BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
    try // must catch exceptions
        int u = Integer.parseInt(in.readLine());
        int[] x = new int[u];
        for (int i = 0; i < x.length; i++)
            x[i] = i;
        System.out.println(array_max(x));
    } catch (IOException e) {
        System.out.println("Invalid input.");
    } catch (NumberFormatException e) {
        System.out.println("Invalid input.");
    }
}
Arrays in Ada

with Text_IO; use Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure ArrTest is -- <> unspecified sized array
  Type IntToInt is array (INTEGER range <>) of INTEGER;
  function array_max(a: IntToInt) return integer is
    temp: integer;
    begin
      temp := a(a'first); -- first subscript value for a
      -- a'range = set of legal subscripts
      for i in a'range loop
        if a(i) > temp then
          temp := a(i);
        end if
      end loop
      return temp;
    end array_max;

    size : integer;
    begin
      put_line("Input a positive integer");
      get(size);
      declare -- we need the size before we declare it
        x : IntToInt(1..size); -- dynamically sized array
        max : integer;
        begin
          for i in x'range loop -- x'range = 1..size
            x(i) := i;
          end loop;
          put(array_max(x));
          new_line;
        end
    end ArrTest
Declaring Multidimensional Arrays

- In C
  int x[10][20];
- In Java
  int[][] x = new int[10][20];
- In Ada
  type Matrix_Type is array(1..10, -10..10) of integer;
  which is different from
  type Matrix_Type is array(1..10) of array (-10..10) of integer;
- In FORTRAN
  INTEGER x(10, 20)

Row Major Matrix

```
int i[4][5];

<table>
<thead>
<tr>
<th></th>
<th>x[0][0]</th>
<th>x[0][1]</th>
<th>x[0][2]</th>
<th>x[0][3]</th>
<th>x[0][4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[1][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[3][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[0][1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[1][1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[2][1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[3][1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[0][2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[1][2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[2][2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[3][2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[0][3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[1][3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[2][3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[3][3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[0][4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[1][4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[2][4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x[3][4]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Column Major Matrix

```
int i[4][5];
```

<table>
<thead>
<tr>
<th></th>
<th>x[0][0]</th>
<th>x[0][1]</th>
<th>x[0][2]</th>
<th>x[0][3]</th>
<th>x[0][4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[1][0]</td>
<td>x[1][1]</td>
<td>x[1][2]</td>
<td>x[1][3]</td>
<td>x[1][4]</td>
<td></td>
</tr>
</tbody>
</table>

Multidimensional Parameters in C

- In C and C++, the size of array is not specified when the formal parameters are declared.
- This is problematic for multidimensional arrays, therefore all but the first dimension must be specified:
  ```c
  int array_max(int a[][20], int size);
  ```
- This allows the compiler to calculate the offsets for a[1], a[2], etc.
Associative Arrays

- An associative array is an unordered collection of data elements that are indexed by an equal number of values called keys.
- Associative arrays are a native data type to Perl and are supported by standard class libraries in C++ and Java.
- An associative array (also called hashes) can be initialized in Perl by writing:
  ```perl
  %salaries = ("Gary" => 75000,
              "Perry" => 57000, "Mary" => 55750,
              "Cedric" => 47850);
  ```

Associative Array Operations

- You can reference an individual element:
  ```perl
  $salaries{"Perry"} = 58850;
  ```
- You can delete an individual element:
  ```perl
  delete $salaries{"Gary"};
  ```
- You can empty the entire hash:
  ```perl
  @salaries = ();
  ```
- The exists operator tells if an element exists with a specific key:
  ```perl
  if (exists $salaries{"Shelly"}) ...
  ```
Implementing Associative Arrays

• Both the key and data value are stored.
• A 32-bit hash value is computed for each entry for fast lookup.
• When the array must be expanded, more bits are added to the hash value and only half the array must be moved.

Records

• Structures (or records) are heterogeneous collections of data items; most programming languages let us create such collections.
• An example of this might be creating a record that contains an integer, a character and a double (or float).
• The most common operator for a field within the record is the period ( . ).
• Most languages use sequential allocation for the components of a record.
Structures in C

```c
struct IntCharReal {
    int i;
    char c;
    double r;
};
struct {
    int j;
    char ch;
    double d;
} w; /* Is this the same as above */
f() {
    struct IntCharReal x;
    x.i = 7;
    /* . is a component selector operator */
}
```

Structures in Pascal

```pascal
TYPE
    IntCharReal = RECORD
        i : Integer;
        c : Char;
        d : Float
    END;
VAR
    x : IntCharReal;
BEGIN
    x.i := 7;
```
Tuples in ML

- ML has a record structure that is closer to Cartesian product:
  \[
  \text{type IntCharReal = int * char * real;}
  \]
- Valid values for this type include \((42, \text{"z"}, 1.1)\) and \((2, \text{"a"}, 3.14)\)
- Its projections are #1, #2 and #3.

Unions

- A union allows two different data types to coexist in the same memory space.
- Unions are discriminated (where there is a tag indicating which type is valid at the moment) or undiscriminated.
Unions in C

    union IntOrReal   {
        int     i;
        double r;
    }

    f()
    {
        IntOrReal u;
        u.i = 5;
        u.r = 4.5;
        printf (%d\n", u.i); /* prints what */
    }

Discriminated Unions in C

    enum Disc {IsInt, IsReal}
    union IntOrReal   {
        enum Disc     which;
        union         {
            int         i;
            double      r;
        } val;
    }

    f()
    {
        IntOrReal x;
        x. which = IsReal;
        x.val.r = 2.3;
        ...
        if (x. which == IsInt)
            printf("%d\n", u.i);
Variant Records In Pascal

PROGRAM EmpStuff;
{********************************************}
{* This program is an illustration of      *}
{* VARIANT records                         *}
{* and is essentially a set of declarations*}
{* and a driver for two procedures:        *}
{* ReadEmployee - Reads in data for a new*}
{* employee record.                        *}
{* PrintEmployee- Prints the record       *}
{********************************************}
CONST
  StringLength = 20;   { Length of all strings}
TYPE
  StringType = String[StringLength];
  IDRange = 1111..9999;
  Gender = (Female, Male);

  MaritalStatus = (Married, Divorced, Single);

  Employee = RECORD
    ID : IDRange;
    Name : StringType;
    Sex : Gender;
    { Variant portion }
    CASE MS : MaritalStatus OF
      Married : (SpouseName : StringType;
        NumberKids : Integer);
      Divorced : (NumKids : Integer);
      Single : (LivesAlone : Boolean)
    END;
  END;

VAR
  ThisGuy : Employee;
PROCEDURE ReadEmployee(VAR Guy : Employee);
{*******************************************************
/* Reads in the employee record from */
/* the keyboard, prompting the user as */
/* necessary. */
{*******************************************************
VAR
  SexCode, LivesAloneChar : Char;
  MS : MaritalStatus;
  Choice : Integer;
BEGIN
  WITH Guy DO
  BEGIN
    Write('ID  ?');
    ReadLn(id);
    Write('Name   ?');
    ReadLn(Name);
    REPEAT
      Write('(M)ale or (F)emale    ?');
      ReadLn(SexCode);
      CASE SexCode OF
        'M', 'm' : Sex := Male;
        'F', 'f' : Sex := Female
      END  { case }
    UNTIL UpCase(SexCode) IN ['F', 'M'];
    {*******************************************************
    /* Determine which is the correct */
    /* "tag" */
    /* This’ll determine which variant */
    /* that we will fill. */
    {*******************************************************
    WriteLn('Type Choice Number');
    WriteLn('1. Married');
    WriteLn('2. Divorced');
    WriteLn('3. Single');
  END  { Guy }
END  { ReadEmployee }
ReadLn(Choice);
MS := MaritalStatus(Choice-1);

{*******************************************************************************
(* We will read data for one and *)
(* only one variant              *)
*******************************************************************************
CASE ms OF
    Married : BEGIN
        Write('Spouse''s name ?');
        ReadLn(SpouseName);
        Write('Number of children ?');
        ReadLn(NumberKids)
    END;  { case married }
    Divorced : BEGIN
        ReadLn(NumKids)
    END;  { case divorce  }

    Single : BEGIN
        REPEAT
            Write('Lives alone (y-n)?');
            ReadLn(LivesAloneChar);
            UNTIL UpCase(LivesAloneChar) IN ['N', 'Y'];

            LivesAlone :=
                UpCase(LivesAloneChar) = 'Y'
        END { case single }
    END { case ms of.. }

END; { ReadEmployee }
PROCEDURE PrintEmployee(Guy : Employee);

{****************************************}
{* Prints the employee record, using *}
{* an a CASE to print the proper gender*}
{****************************************}
VAR MS : MaritalStatus;
BEGIN
  WITH Guy DO
    BEGIN
      WriteLn('ID: ', id);
      WriteLn('Name: ', Name);
      Write('Sex: ');
      CASE Sex OF
        Male   : WriteLn('Male');
        Female : WriteLn('Female');
      END;  { case }
      CASE MS OF
        Married : BEGIN
          WriteLn('The spouse''s name is ',
            SpouseName);
          WriteLn('They have ',
            NumberKids:1,
            ' children.'
            END;  { case married }
        Divorced : WriteLn('S)he has ',
            NumKids:1, ' children.');
        Single : IF LivesAlone
                  THEN WriteLn('Lives alone.')
                  ELSE WriteLn
                    ('Doesnt live alone.')
        END  { case ms of.. }
    END;  { with Guy.. }
END;  { PrintEmployee }
BEGIN { EmployeeStuff }
{**********************************************************}
(* Read the new employee record, *)
(* skip 2 lines and print it. *)
{**********************************************************}
  ReadEmployee(ThisGuy);
  WriteLn;
  WriteLn;
  PrintEmployee(ThisGuy);
END. { EmployeeStuff }

Unions in ML

- The syntax used for enumerations
  
  ```ml
  datatype Color_type = Red | Green | Blue;
  ```

- is extended to unions:

  ```ml
  datatype ntOrReal = IsInt of int | IsReal of real;
  ```

- IsInt and IsReal are tags that we can use to determine the type of the value being stored:

  ```ml
  val x = IsReal(2.3)
  ```
Pointers

• A pointer is an item whose stored value is a reference to another item.
• In C
  \[\text{int } \ast p;\]
  allocates storage for the pointer p, not the integer to which it is pointing.
• This pointer may be undefined pointing to an arbitrary location.

Pointer Types

• A pointer type is one in which the variables have a range of values that consist of memory addresses and a special value (called \textit{nil} or sometimes \textit{null}).
• Pointers serve two main purposes:
  – They provide some of the power of indirect addressing
  – They provide a mechanism for dynamic storage management, using \textit{heap-dynamic variables}.
• While pointers are not structured variables, they are defined using a type operator (* in C/C++ and \textit{access} in Ada).
**Pointer Design Issues**

- What are the scope and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable?
- Are pointers restricted to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect address, or both?
- Should the language support pointer types, references types or both?

**Dangling Pointers**

- A dangling pointer is a pointer that contains the address of a heap-dynamic variable that has been deallocated.
- Dangling pointers are dangerous because:
  - the location may have been allocated to a new heap-dynamic variable whose type may be different.
  - the location may be in use by the storage management and changing its value may compromise the entire storage manager.
Lost Heap-Dynamic Variables

- A lost heap-dynamic variable is an allocated heap-dynamic variable that is no longer accessible to the user program. They are also known as garbage.
- This happens by:
  - Pointer p1 being set to point to a newly create heap-dynamic variable.
  - p1 is later set to point to another newly create heap-dynamic variable.
- This leads to a problem known as memory leakage.

Pointers In C

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p = NULL;
    p = (int *) malloc(sizeof(p));
    if (p != NULL) *p = 333;
    printf("p points to %d\n", *p);
    free(p);
    return(0);
}
```
Pointers In C++

#include <iostream.h>

int main(void)
{
    int *p = new int;

    *p = 333;
    cout << *p << endl;
    delete p;
    return(0);
}

Pointers In Pascal

PROGRAM PTest;
  TYPE
    IntPtr = ^Integer;
  VAR
    p : IntPtr;
  BEGIN
    New(p);
    IF p = NIL
      THEN WriteLn('P isn't pointing anywhere.');
    ELSE BEGIN
       p^ := 333;
       WriteLn('P points to ', p^);
       Dispose(p)
    END { else }
  END. { Ptest }
Pointers In Java

// in Java must use class Integer, not int
// also, cannot allocate without assigning
// a value

Integer x = new Integer(2);
// print the actual integer value of x, ie 2

System.out.println(x);
// no delete operation allowed

Pointers In Ada

declare
  // access means pointer
type Intptr is access Integer;
procedure delete_int is
  new Ada.Unchecked_Deallocation(Integer,
      Intptr);

  x : Intptr := new Integer;
begin
  x.all := 2;  -- in C would be *X = 2
  put(x.all); new_line;
  delete_int(x);
end;