Data Types

CS 315 – Programming Languages
Pinar Duygulu
Bilkent University

Introduction

- A *data type* defines a collection of data objects and a set of predefined operations on those objects
- In the early languages there were only a few data structures (for example linked lists and binary trees were modeled by arrays)
- COBOL allowed programmers to specify accuracy of decimal data values
- In ALGOL 68 the programmer allowed to design the needed types – user defined types
- Then abstract data types are introduced starting from Ada 83 – the use of a type is separated from the representation and a set of operations on values of that type

Introduction

- A *descriptor* is the collection of the attributes of a variable – collection of memory cells that store variable attributes
- If the attributes are static descriptors are required only in compile time
- For dynamic attributes part or all of the descriptor must be maintained during execution

- An *object* represents an instance of a user-defined (abstract data) type

- One design issue for all data types: What operations are defined and how are they specified?

Primitive Data Types

- The data types that are not defined in terms of other types are called **primitive data types**.
  - They are mostly reflections of hardware, or require little hardware support.
  - They are used to provide the structured types.

- Almost all programming languages provide a set of *primitive data types*

Primitive Data Types

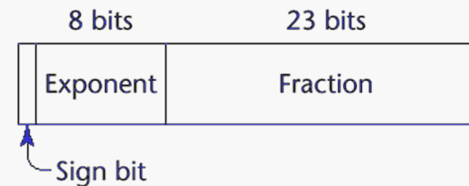
- Numeric Types : Integer, Floating-point, decimal
- Character types
- Boolean Types

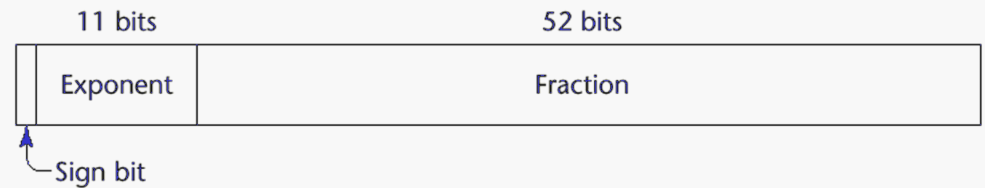
Primitive Data Types: Integer

- The most common primitive numeric data type is **integer**.
- Almost always an exact reflection of the hardware so the mapping is trivial
- Most computers supports several sizes of integer (short, long, unsigned, …)
- There may be as many as eight different integer types in a language
- Java's signed integer sizes: byte, short, int, long
- C++ and C# include unsigned integer types
- Most computers use **twos complement** to store negative values.

Primitive Data Types: Floating Point

- Model real numbers, but only as approximations
- In general, infinite amount of space is needed to represent a real value.
- This causes **loss of accuracy** on real valued operations.
- Usually represented as **fractions (mantissa)** and **exponents**.
- Languages for scientific use support at least two floating-point types (e.g., `float` and `double`; sometimes more
- Usually exactly like the hardware, but not always
- IEEE Floating-Point Standard 754





Primitive Data Types: Decimal

- Fixed number of decimal digits, with the decimal point at a fixed position.
- Primary data type for business applications, essential to COBOL
  - C# offers a decimal data type
- Stored by using binary codes for the decimal digits.
- It takes 4 bits to code a decimal digit.
- *How ?*
- It takes 24 bits to store a 6 digit number in BCD (Binary Coded Decimal), 20 bits in binary.

- *Advantage*: accuracy
- *Disadvantages*: limited range, wastes memory

Primitive Data Types: Character

- Stored as numeric codings
- Most commonly used coding: ASCII
  - Values 0..127 are used to code 128 characters.
- An alternative, 16-bit coding: Unicode
  - 16 bits. \u0000 … \uFFFF
  - Includes characters from most natural languages
  - Originally used in Java
  - C# and JavaScript also support Unicode

Character String Types

- Values are sequences of characters
- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Should the length of strings be static or dynamic?

Strings and Their Operations

- Typical operations:
  - Assignment and copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching

Strings and Their Operations

- In Pascal, C, C++, and Ada strings are stored in arrays of single char.
  - Example:

    Pascal: `var MONTH : packed array [1..9] of char;`
    `MONTH = 'November '`

- In Ada `STRING` is a predefined type as a single-dimensioned array of `CHARACTERS`.
  - `NAME(3:5)` is a <u>substring</u> of the string `NAME`.
  - Character string <u>concatenation</u> operation in Ada is specified by `&`
    - `NAME1 := NAME1 & NAME2;`

Strings and Their Operations

- C and C++ : not primitive
  - use char arrays to store character strings

    ```
    char *str = "apples";
    ```
  - A string is terminated by a special character (null)
  - a library of functions that provide operations : strcmp, strcpy, strcat, strlen
- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including elaborate pattern matching
- Java
  - Primitive via the `String` class

Character String Length Options

- Static:
  - FORTRAN77 - 90, COBOL, Pascal
  - `CHARACTER (LEN=15) NAME1, NAME2`
  - Java's `String` class
- *Limited Dynamic Length*: C and C++
  - In C-based language, a special character is used to indicate the end of a string's characters, rather than maintaining the length
- *Dynamic* (no maximum): SNOBOL4, Perl, JavaScript
- Ada supports all three string length options

Character String Type Evaluation

- Aid to writability
- As a primitive type with static length, they are inexpensive to provide--why not have them?
- Dynamic length is nice, but is it worth the expense?

Character String Implementation

- Static length: compile-time descriptor
- Limited dynamic length: may need a run-time descriptor for length (but not in C and C++)
- Dynamic length: need run-time descriptor; allocation/de-allocation is the biggest implementation problem

Static string
Length
Address

Compile-time descriptor for static strings

Limited dynamic string
Maximum length
Current length
Address

Run-time descriptor for limited dynamic strings

User defined Ordinal Types

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
- Examples of **primitive ordinal types**: integer, char, boolean
- Examples of **user defined ordinal types**: enumeration and subrange.

Enumeration Types

- All possible values are **enumerated in the definition**.
- Values are **symbolic constants**.
- Common operations: predecessor, successor, position in the list, and the value for a given position number.
- C# example

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

Enumeration Types

Example in C:

```
/* enumeration type in C */
main () {
  enum day {mo, tu, we, th, fr, sa, su};
  enum day today;
  today = th;
  if ((today == sa) || (today == su))
   printf ("weekend\n");
  else printf ("weekday\n");
}
```

Enumeration Types

- Design issues
  - Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant checked?
    - No, in Pascal, C and C++.
    - Yes, in Ada. Such literals are called overloaded **literals**.
  - Are enumeration values coerced to integer?
  - Any other type coerced to an enumeration type?

Enumerated Types: Evaluation

- Aid to readability, e.g., no need to code a color as a number
- Aid to reliability, e.g., compiler can check:
  - operations (don't allow colors to be added)
  - No enumeration variable can be assigned a value outside its defined range
  - Ada, C#, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types

Subrange Types

- An ordered contiguous subsequence of an ordinal type
  - Example: 12..18 is a subrange of integer type
- Ada’s design

```
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;

Day1: Days;
Day2: Weekday;
Day2 := Day1;
```

Evaluation of Subrange Types

- Aid to readability
  - Make it clear to the readers that variables of subrange can store only certain range of values
- Reliability
  - Assigning a value to a subrange variable that is outside the specified range is detected as an error

Implementation of User Defined Ordinal Types

- Enumeration types are implemented as integers
- Subrange types are implemented like the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

Array Types

- An array is an aggregate of homogeneous data elements in which an individual element is identified by its position in the aggregate, relative to the first element.

Array Design Issues

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- What is the maximum number of subscripts?
- Can arrays be initialized when their storage is allocated?
- What kind of slices are allowed, if any?

Array Indexing

- *Indexing* (or subscripting) is a mapping from indices to elements

  `array_name (index_value_list) → an element`

- Index Syntax
  - FORTRAN, PL/I, Ada use parentheses
    - `SUM = SUM + A(I)`
    - In FORTRAN, compiler must determine by matching the name to function and array declarations.
    - Not readable.
  - Most other languages use brackets

Array Index (Subscript) Type

- FORTRAN, C: integer only
- Pascal: any ordinal type (integer, Boolean, char, enumeration)
- Ada: integer or enumeration (includes Boolean and char)
- Java: integer types only

- C, C++, Perl, and Fortran do not specify range checking
- Java, ML, C# specify range checking

Subscript Binding and Array Categories

- There are four categories of arrays
- Static
- Fixed stack dynamic
- Stack dynamic
- Fixed Heap Dynamic

Subscript Binding and Array Categories

- ***Static:*** subscript ranges are statically bound and storage allocation is static (before run-time)
  - Advantage: efficiency (no dynamic allocation)
  - Example:
  - FORTRAN77 arrays.
  - Subscript type is `INTEGER` (design time binding).

Subscript Binding and Array Categories

- *Fixed stack-dynamic*: subscript ranges are statically bound, but the allocation is done at declaration time
  - Advantage: space efficiency
  - Example:
  - Arrays declared in Pascal and C subprograms.

Subscript Binding and Array Categories

- *Stack-dynamic*: subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)
- Once subscript ranges and storage is allocated, they remain fixed during the lifetime of the variable.
- Advantage: flexibility (the size of an array need not be known until the array is to be used)
- Example:
- **Conformant arrays** in ISO Standard Pascal.

Subscript Binding and Array Categories

```
Program Conformant;
var alist: array[1..10] of integer;
  blist: array[-20..100] of integer;
  sum: integer;
procedure sumlist (var sum:  integer;
                       list: array[lower..upper: integer]
                         of integer);
   var index: integer;
begin
   sum := 0;
   for index := lower to upper do
       sum := sum + list[index];
end; {sumlist}

begin {Conformant}
  . . .
  sumlist(sum, alist);
  written(sum);
  sumlist(sum, blist);
  written(sum);
end. {Conformant}
```

Subscript Binding and Array Categories

- *Fixed heap-dynamic*: similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation (i.e., binding is done when requested and storage is allocated from heap, not stack)

Subscript Binding and Array Categories

- ***Heap-dynamic:*** binding of subscript ranges and storage allocation is dynamic and can change any number of times
  - Advantage: flexibility (arrays can grow or shrink during program execution)
  - Example:
  - Arrays in APL and LISP.

Subscript Binding and Array Categories

- C and C++ arrays that include `static` modifier are static
- C and C++ arrays without `static` modifier are fixed stack-dynamic
- Ada arrays can be stack-dynamic
- C and C++ provide fixed heap-dynamic arrays
- C# includes a second array class `ArrayList` that provides fixed heap-dynamic
- Perl and JavaScript support heap-dynamic arrays

The Number of Subscripts in Arrays

- Early versions of FORTRAN limited the number of array dimensions to 3, then 7.
- Most contemporary languages do not have such limitations.
- Arrays in C can have only one subscript, but an element can be another array.

Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example

```
int list [] = {4, 5, 7, 83}
float x[] = {3, 5.5, 0.1, 1}; size=4
```

  - Character strings in C and C++

```
char name [] = "freddie";
```

  - Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", "Joe"];
```

  - Java initialization of String objects

```
String[] names = {"Bob", "Jake", "Joe"};
```

Array Initialization

FORTRAN77 example:

```
      dimension x(2:5)      x(2), x(3), x(4), x(5)
      integer a(4)          a(1), a(2), a(3), a(4)
      data x /3, 5.5, 0.1, 1/
      data a / 3*0, 7/        0,0,0,7
      do 10 i=2,5
      print *, "X(",i,")= ", x(i)
   10 continue
      do 20 i=1,4
      write(6,100) i, A(i)
   20 continue
  100 format('A(',I1,')=',I2)
      end
```

Array Initialization

Compiler sets the size of the array. But, cannot detect a possible error, due to typing wrong number of values.

Ada also allows array initialization.

Pascal and Modula-2 do not allow array initialization.

Arrays Operations

- An **array operation** is one that operates on an array as a unit.
- APL provides the most powerful array processing operations for vectors and matrixes as well as unary operators (for example, to reverse column elements)
- Ada allows array assignment but also catenation
- Fortran provides *elemental* operations because they are between pairs of array elements
  - For example, + operator between two arrays results in an array of the sums of the element pairs of the two arrays

Rectangular and Jagged Arrays

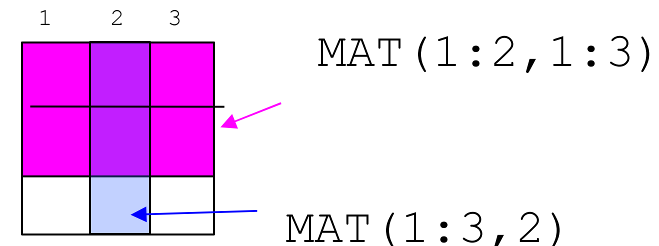
- A rectangular array is a multi-dimensioned array in which all of the rows have the same number of elements and all columns have the same number of elements
- A jagged matrix has rows with varying number of elements
  - Possible when multi-dimensioned arrays actually appear as arrays of arrays

Slices

- A slice is some substructure of an array;
- A mechanism for referencing part of an array as a unit.
- Slices are only useful in languages that have array operations

Slices

- FORTRAN90 examples
- `1 2 3`INTEGER VECTOR(1:10), MAT(1:3, 1:3), CUBE(1:3, 1:3, 1:4)
- `MAT(1:2,1:3)VECTOR(3:6)` is a four-element array
- `MAT(1:3,2)` refers to the second column of `MAT`
- `MAT(1:3,2)MAT = CUBE(1:3, 1:3, 2)` is legal. Also
- `CUBE(1:3, 1:3, 2) = MAT` is legal.
- `VECTOR((/3, 2, 1, 8/))` is a vector formed by taking 3rd, 2nd, 1st, and 8th elements of `VECTOR`.

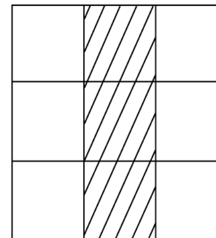


Slices

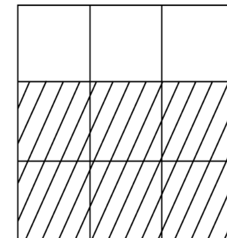
- Fortran 95

```
Integer, Dimension
(10) :: Vector

Integer, Dimension
(3, 3) :: Mat

Integer, Dimension
(3, 3) :: Cube
```

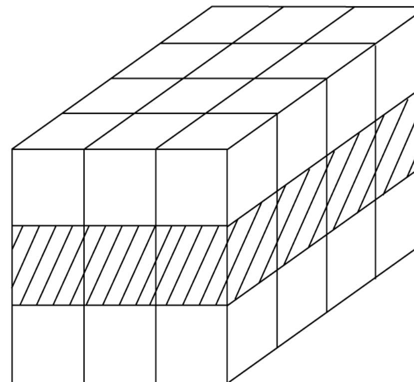
`Vector (3:6)` is a four element array



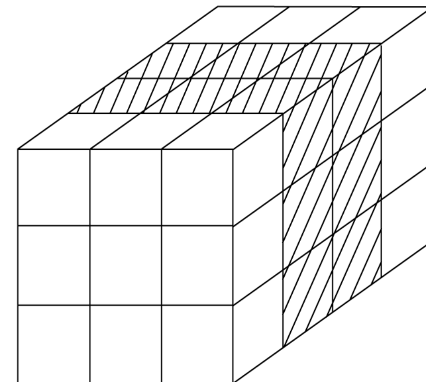
MAT (1:3, 2)



MAT (2:3, 1:3)



CUBE (2, 1:3, 1:4)



CUBE (1:3, 1:3, 2:3)

Implementation of Arrays

- The code to allow accessing of array elements must be generated at **compile time**.
- At **run time**, this code is executed.

- Access function maps subscript expressions to an address in the array
- Access function for single-dimensioned arrays:

  address(list[k]) = address (list[lower_bound])
  + ((k-lower_bound) * element_size)

Implementation of Arrays

`list[k]`

The ***access function*** would be:

address(`list[k]`)=address(`list[`lb`]`)+(`k`-lb)*element_size

address(`list[k]`)=(address(`list[`lb`]`)-(lb*elt_size))+
(`k`* elt_size)

lb: Index lower bound

elt_size : element size

lb*elt_size : *Constant, if the base address is known at compile time*

Implementation of Arrays

- If the element type is statically bound and the array is statically bound, then the value of the constant part can be computed before run time.
- Variable part is computed and added in the run time.

Accessing Multi-dimensioned Arrays

- Hardware memory is simple list of bytes (single dimensional array).
- Multidimensional arrays must be mapped to a single dimension.
- For example,
- 1 2 3
- 4 5 6
- 7 8 9
-
- Two common ways:
  - Row major order (by rows) – used in most languages
  - 1,2,3,4,5,6,7,8,9
  - Three dimensions: A[1,1,1], A[1,1,2],A[1,1,3],A[1,2,1]…
  - column major order (by columns) – used in Fortran
  - 1,4,7,2,5,8,3,6,9
  - Three dimensions:
  - A[1,1,1], A[2,1,1],A[3,1,1],A[1,2,1]…

Accessing Multi-dimensioned Arrays

- **Row major order**: elements with lower value in the first subscript are stored earlier. Then, elements with lower value in the second subscript are stored, so on.
- **Column major order**: elements with lower value in the last subscript are stored earlier. Then, elements with second value in the last subscript are stored, so on.
- **FORTRAN** uses **column major order**, the other languages use row major order.

Accessing Multi-dimensioned Arrays

- **Access function**: mapping of the base address of an array and the set of index values to the addresses in the memory.
- **Access function** for **2-dimensional array** stored in **row major order:**
- base address + (element_size * number of elements preceeding)
- address(a[i,j]) = address(a[row_lb, col_lb]) +
- ((i-row_lb) * *n* + (j-col_lb)) * element_size
- *n* is the number of elements per row. This can be rearranged:
- address (a[i,j]) = address(a[row_lb, col_lb])
- \- (row_lb * *n*+ col_lb) * element_size
- \+ (i * *n* + j) * element_size
- Here, the first two terms are constant, the last is variable.

Accessing Multi-dimensioned Arrays

- General format
  Location (a[I,j]) = address of a [row_lb,col_lb] + (((I – row_lb) * n) + (j – col_lb)) * element_size

Compile time Descriptors

Array
Element type
Index type
Index lower bound
Index upper bound
Address

Single-dimensioned array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
⋮
Index range *n*
Address

Multi-dimensional array

Associative Arrays

- An *associative array* is an unordered collection of data elements that are indexed by an equal number of values called *keys*
  - User defined keys must be stored
- Design issues: What is the form of references to elements

Associative Arrays

- Names begin with `%`; literals are delimited by parentheses

```
%hi_temps = ("Mon" => 77, "Tue" => 79, "Wed" => 65, …);
```

- Subscripting is done using braces and keys

```
$hi_temps{"Wed"} = 83;
```

  – Elements can be removed with `delete`

```
delete $hi_temps{"Tue"};
```

Associative Arrays

Provided in Perl (called hash) and Java.

- Indexed by keys
- The size can grow and shrink dynamically.

```
#!/usr/local/bin/perl
%notlar = (“Ali” => 95, “Veli” => 80, “Selami” => 95”);
```

To print an entry:

```
print “Veli’nin notu $notlar{\“Veli\”}\n”;
```

To add a new entry:

```
$notlar{“Hulki”} = 75;
```

To delete an entry:

```
delete $notlar{“Veli”};
```

To change an entry:

```
$notlar{“Veli”} = 82;
```

To go through the array:

```
foreach $anahtar (keys %notlar) {
  print “$anahtar $notlar{\”$anahtar\”}\n”;
}
```

To empty te array:

```
%notlar = ();
```

Record Types

- A *record* is
  - a possibly heterogeneous aggregate of data elements
  - in which the individual elements are identified by names
- Design issues:
  - What is the syntactic form of references to the field?
  - Are elliptical references allowed

Definition of Records

- COBOL uses level numbers to show nested records; others use recursive definition
- Record Field References

  1. COBOL

  field_name OF record_name_1 OF ... OF record_name_n

  2. Others (dot notation)

  record_name_1.record_name_2. ... record_name_n.field_name

Definition of Records in COBOL

- COBOL uses level numbers to show nested records; others use recursive definition

```
01 EMP-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID   PIC X(10).
      05 LAST  PIC X(20).
   02 HOURLY-RATE PIC 99V99.
```

Definition of Records in Ada

- Record structures are indicated in an orthogonal way

```
type Emp_Rec_Type is record
First: String (1..20);
Mid: String (1..10);
Last: String (1..20);
Hourly_Rate: Float;
end record;
Emp_Rec: Emp_Rec_Type;
```

References to Record Fields

- Most language use dot notation
  `Emp_Rec.Name`
- Fully qualified references must include all record names
- Elliptical references allow leaving out record names as long as the reference is unambiguous, for example in COBOL

`FIRST`, `FIRST OF EMP-NAME`, and `FIRST` of EMP-REC are elliptical references to the employee's first name

References to Record Fields

- employee.name.last *fully qualified reference*
- with employee do begin
- …
- … name.last … *eliptical reference*
- … *must be unambiguous*
- end

Operations on Records

- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides `MOVE CORRESPONDING`
  - Copies a field of the source record to the corresponding field in the target record

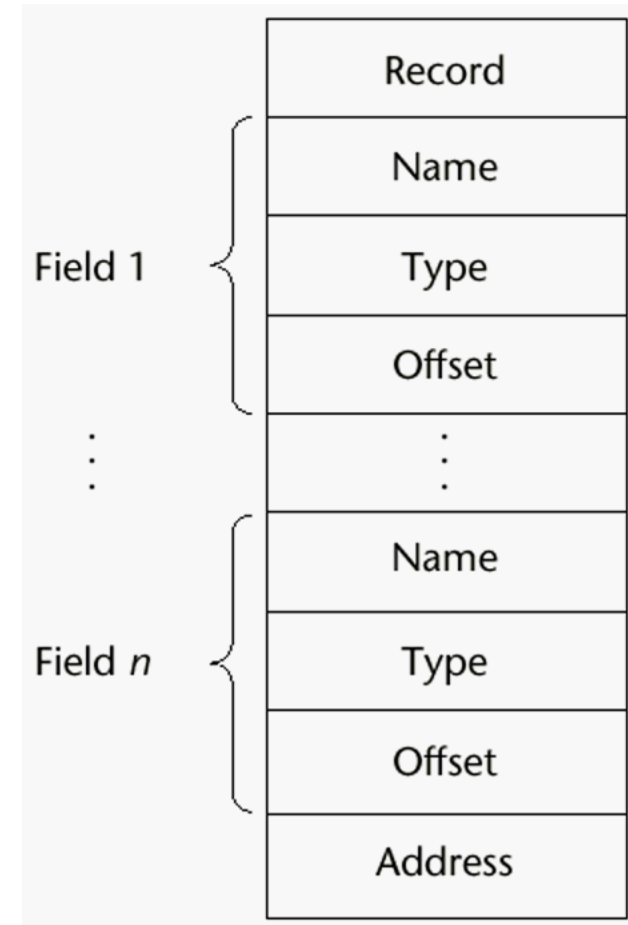
- In Pascal and Modula-2: can be assigned, but no other operations
- Ada: assigment, comparison (equality, inequality)
- C: address-of
- ANSI C, C++: assignment

Evaluation and Comparison to Arrays

- Straight forward and safe design
- Records are used when collection of data values is heterogeneous
- Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
- Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

Implementation of Record Type

Offset address relative to the beginning of the records is associated with each field



Union Types

- A *union* is a type whose variables are allowed to store different type values at different times during execution
- Design issues
  - Should type checking be required?
  - Should unions be embedded in records?

Discriminated vs. Free Unions

- Fortran, C, and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called *free union*
- Type checking of unions require that each union include a type indicator called a *discriminant*
  - Supported by Ada

Free Unions

- FORTRAN, C, C++ provide free unions.
- No language support for type checking provided.
- Programmer has the complete freedom from type checking.

The Discriminated Unions of ALGOL 68

- Type checking of unions requires that each union construct include a type indicator.
- Such indicator is called a **tag**, or **discriminant**.

Pascal Union Types

Discriminated union structure: **Variant records,**

It may be a **part of a record**.

Problems:

User can change the tag field.
One can access the variant part without checking the current value of the tag field.

The tag field can be omitted.

```
type shape = (circle, triangle, rectangle);
   object = record case form: shape of
            circle: (diameter: real);
            triangle: (left, right: integer;
                       angle: real);
            rectangle: (side1, sidde2: integer)
         end;
```

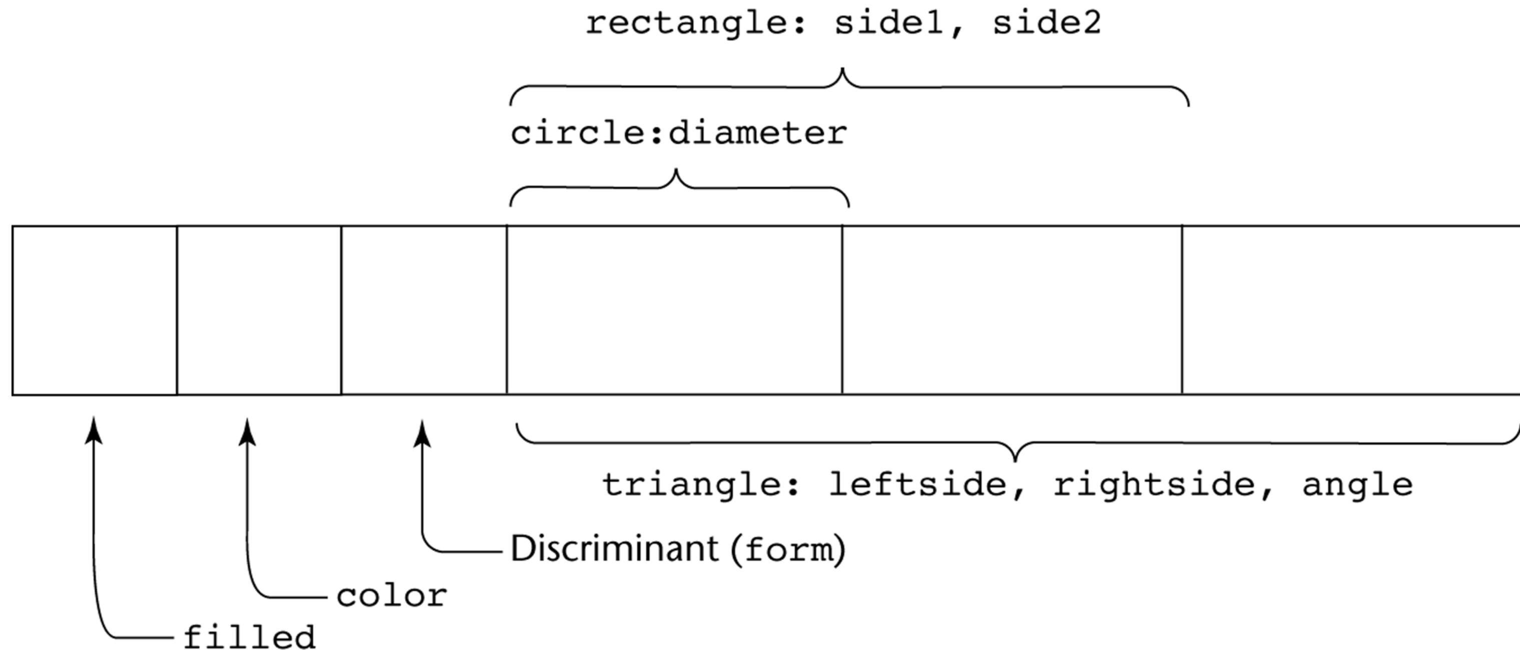
Tag field can be omitted:

```
type shape = (circle, triangle, rectangle);
   object = record case shape of
            circle: (diameter: real);
               . . .
         end;
```

Ada Union Types

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
Filled: Boolean;
Color: Colors;
case Form is
when Circle => Diameter: Float;
when Triangle =>
Leftside, Rightside: Integer;
Angle: Float;
when Rectangle => Side1, Side2: Integer;
end case;
```

Ada Union Type Illustrated



A discriminated union of three shape variables

Evaluation of Unions

- Potentially unsafe construct
  - Do not allow type checking
- Java and C# do not support unions
  - Reflective of growing concerns for safety in programming language

* Set Types

- A **set type** is an unordered collection of distinct values.
- Usually stored as bit strings.
- **type** Digits = set of 0..9;
- CharSet = set of Char;
- var d1: Digits;
- d1 := [7,2,1] is represented by 0110000100.

Sets in Pascal and Modula-2

- **Type** colors = (red, blue, green, yellow);
- Colorset = **set of** colors;
- **Var** set1, set2: colorset;

Pointer and Reference Types

- A **pointer type** is either an address of memory or a special value called nil.
- Provide the power of indirect addressing
- Provide a way to manage dynamic memory
- A pointer can be used to access a dynamically allocated storage, which is called **heap**.
- **Heap-Dynamic variable:** A variable with dynamically allocated storage.
- Such variables have no names, and can only be accessed by pointers. Nameless variables are called **anonymous variables**.
- Pointers are not structured types.

Design Issues for pointers

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

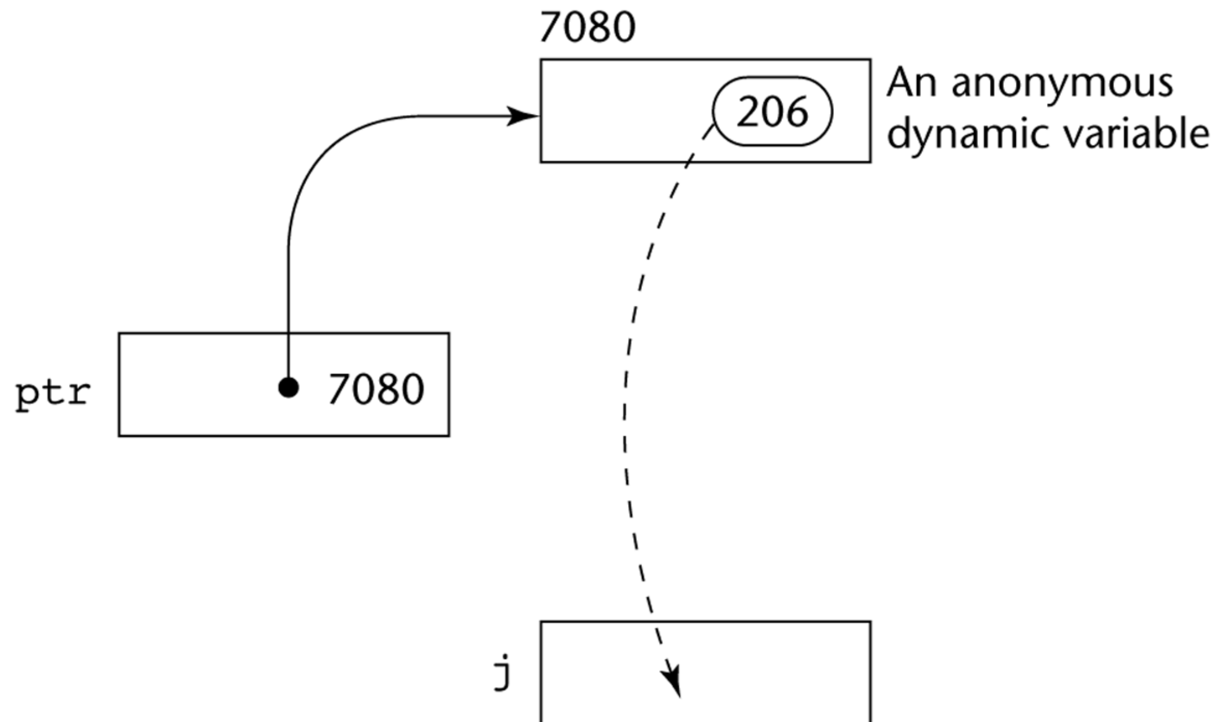
Pointer Operations

- Two fundamental operations: assignment and dereferencing
- Assignment is used to set a pointer variable's value to some useful address
- Dereferencing yields the value stored at the location represented by the pointer's value
  - Dereferencing can be explicit or implicit
  - C++ uses an explicit operation via *

```
j = *ptr
```

sets j to the value located at `ptr`

Pointer Assignment Illustrated



The assignment operation j = *ptr

Pointer Assignment Illustrated

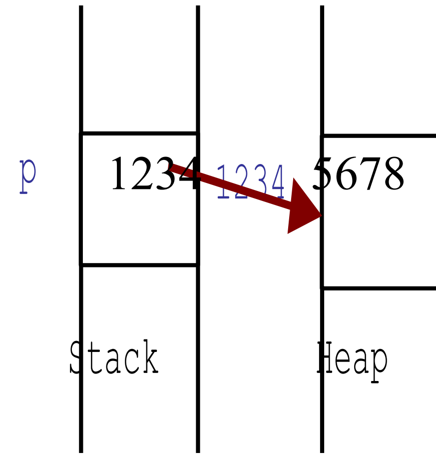
Example:

Normal reference to p yields 1234,

whereas dereferenced reference to p yields 5678.

Pascal: p is 1234, p^ is 5678

In C, and C++, p is 1234, *p is 5678



Problems with pointers

- Dangling pointers
- Lost objects

Dangling Pointers

- A pointer points to a heap-dynamic variable that has been de-allocated (a pointer that contains the address of a dynamic variable that has been deallocated)

Explicit deallocation

through the free or dispose statement.

Example in Pascal:

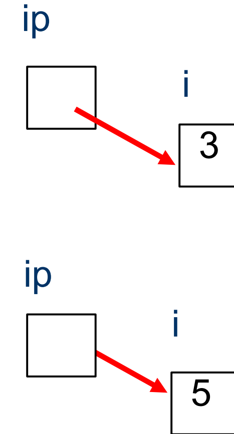
```
Program dangle;
var p,q,r: ^integer;
begin
  new(p);
  p^ := 5;
  q := p;
  dispose(q); {now, q is nil, p is dangling}
  new(r);
  r^ := 7;
  writeln(p^); {prints 7, although p^ := 5}
end.
```

Implicit deallocation

scope of the pointer is larger than the object it points to.

Example in C:

```
int *ip;
void foo() {
  int i=3;
  ip = &i;
} /* foo */
void moo() {
  int j=5;
} /* moo */
main() {
  foo();
  moo();
  /* now ip is a dangling pointer */
  printf("*ip is %d\n", *ip);
  /* prints 5 */
}
```



Dangling pointers

- Dangling pointers are dangerous because we don't get an error message and the program continues execution on wrong data.

Lost Objects

- **Lost object**: is an allocated dynamic object that is no longer accessible to the user program. They are called **garbage**.
- Most often created by the following sequence of actions:
- Pointer p1 is set to point a newly created heap-dynamic variable
- p1 is later set to point to another newly created heap-dynamic variable.
- This problem is called memory leakage.

Pointers in Pascal

- Pointers in Pascal are used only to access dynamically allocated anonymous variables.
- **New**(p) / **dispose**(p) are used to allocate / deallocate a pointer p.
- **Explicit deallocation is costly**. A dynamic variable can be disposed only if no other pointer is pointing to it. **Alternatives**:
- simply ignore the dispose statements.
- Do not include the dispose statement in the language, at all.
- Deallocate the storage, possibly creating dangling pointers
- Implement the dispose correctly.

Pointers in Ada

- Some dangling pointers are disallowed because dynamic objects can be automatically de-allocated at the end of pointer's type scope
- The lost heap-dynamic variable problem is not eliminated by Ada

Pointers in C and C++

- Extremely flexible but must be used with care
- Pointers can point at any variable regardless of when it was allocated
- Used for dynamic storage management and addressing
- Pointer arithmetic is possible
- Explicit dereferencing and address-of operators
- Domain type need not be fixed (**`void *`**)
- `void *` can point to any type and can be type checked (cannot be de-referenced)

Pointer Arithmetic in C and C++

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` is equivalent to `stuff[5]` and `p[5]`

`*(p+i)` is equivalent to `stuff[i]` and `p[i]`

Pointers in Fortran

- Pointers point to heap and non-heap variables
- Implicit dereferencing
- Pointers can only point to variables that have the `TARGET` attribute
- The `TARGET` attribute is assigned in the declaration:

```
INTEGER, TARGET :: NODE
```

Reference Types

- C++ includes a special kind of pointer type called a *reference type* that is used primarily for formal parameters
  - Advantages of both pass-by-reference and pass-by-value
- Java extends C++'s reference variables and allows them to replace pointers entirely
  - References refer to call instances
- C# includes both the references of Java and the pointers of C++

Evaluation of Pointers

- Dangling pointers and dangling objects are problems as is heap management
- Pointers are like `goto`'s--they widen the range of cells that can be accessed by a variable
- Pointers or references are necessary for dynamic data structures--so we can't design a language without them

Representation of Pointers

- Depending on the address space of the machine, pointers are single values stored in either two- or four- byte memory cells.

- Large computers use single values
- Intel microprocessors use segment and offset

Dangling Pointer Problem

- *Tombstone*: extra heap cell that is a pointer to the heap-dynamic variable
  - The actual pointer variable points only at tombstones
  - When heap-dynamic variable de-allocated, tombstone remains but set to nil
  - Costly in time and space

. *Locks-and-keys*: Pointer values are represented as (key, address) pairs
  - Heap-dynamic variables are represented as variable plus cell for integer lock value
  - When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer

Heap Management

- A dynamic variable variable can be pointed to by several pointers.
- It is difficult to determine when such a variable is useless.
- There are two distinct process to reclaim memory:
- **Eager approach**: use **reference counters**, if the reference counter of a variable is 0 then return the space to heap.
- **Lazy approach**: also called **garbage collection**. Run-time system continues to allocate storage cells as requested until all cells are allocated. Then it the **garbage collection** process starts. Associated with each cell there is an indicator. In the beginning of garbage collection they are set to garbage. Then all poiters are traced. The cells pointed by a pointer are set to not garbage. Then all those that are still garbage are reclaimed.
- **Lisp** and **Java** use **automatic garbage collection**.

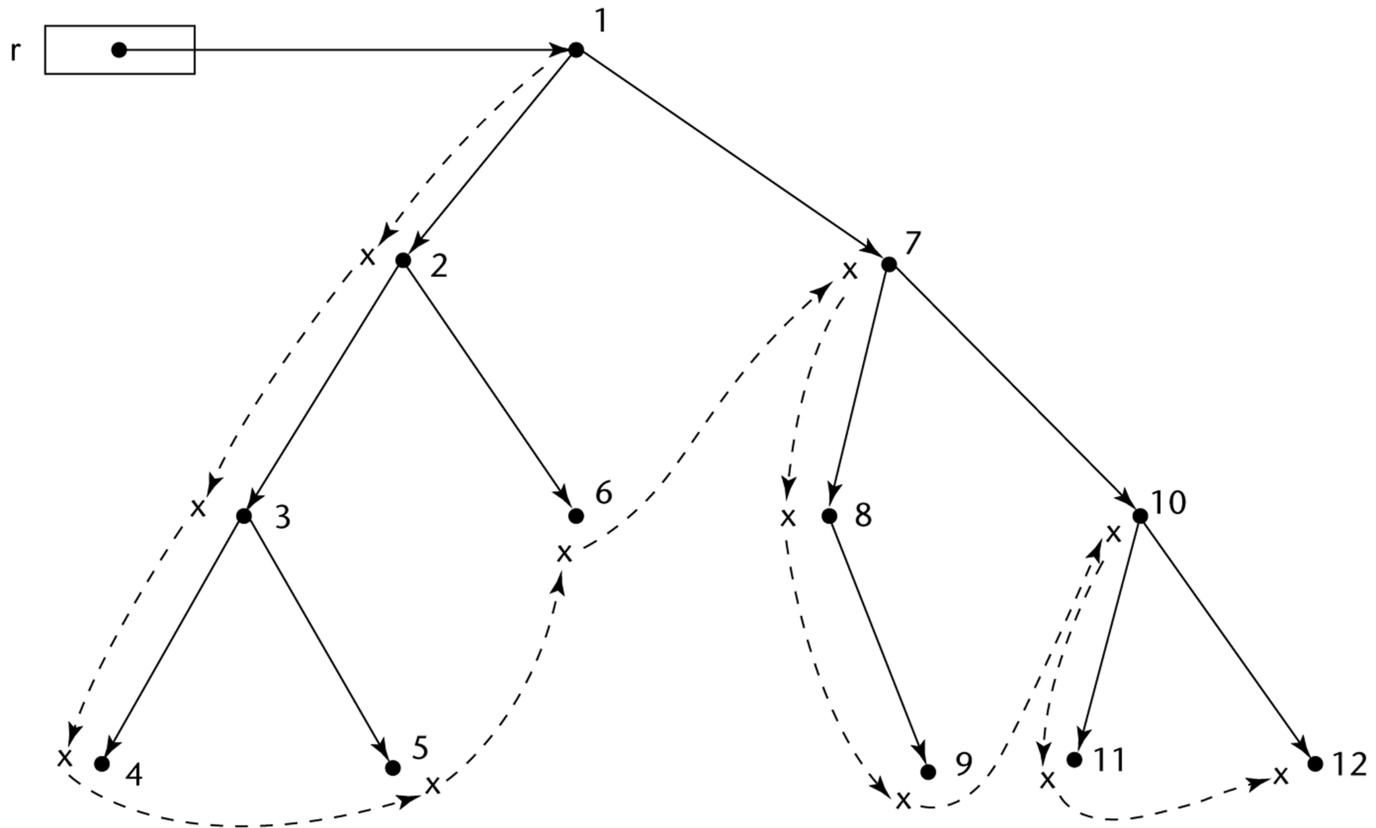
Reference Counter

- Reference counters: maintain a counter in every cell that store the number of pointers currently pointing at the cell
  - Disadvantages: space required, execution time required, complications for cells connected circularly

Garbage Collection

- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; garbage collection then begins
  - Every heap cell has an extra bit used by collection algorithm
  - All cells initially set to garbage
  - All pointers traced into heap, and reachable cells marked as not garbage
  - All garbage cells returned to list of available cells
  - Disadvantages: when you need it most, it works worst (takes most time when program needs most of cells in heap)

Marking Algoritm



Dashed lines show the order of node_marking

Variable-Size Cells

- All the difficulties of single-size cells plus more
- Required by most programming languages
- If garbage collection is used, additional problems occur
  - The initial setting of the indicators of all cells in the heap is difficult
  - The marking process in nontrivial
  - Maintaining the list of available space is another source of overhead

Summary

- The data types of a language are a large part of what determines that language's style and usefulness
- The primitive data types of most imperative languages include numeric, character, and Boolean types
- The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs
- Arrays and records are included in most languages
- Pointers are used for addressing flexibility and to control dynamic storage management