CAPÍTULO 6: DATA TYPES TIPOS DE DATOS

Estructura de los lenguajes

Dr. Christian von Lücken

Capítulo 6 Tópicos

- Introducción
- Tipos de datos primitivos
- Tipos de cadenas de caracteres
- Tipos ordinales definidos por el usuario (User-Defined Ordinal Types)
- Tipos de arrays
- Arrays asociativos
- Registros
- Uniones
- Punteros y tipos referencia

Introducción

- Un tipo de dato define una colección de objetos de datos y un conjunto de operaciones predefinidas sobre estos objetos
- Un descriptor es la selección de los atributos de una variable
- Un objeto representa una instancia de un tipo definido por el usuario (dato abstracto)
- Una cuestión de diseño para todos los tipos es: Qué operaciones son definidas y cómo estas son especificadas?

Tipos de datos primitivos
Primitive Data Types

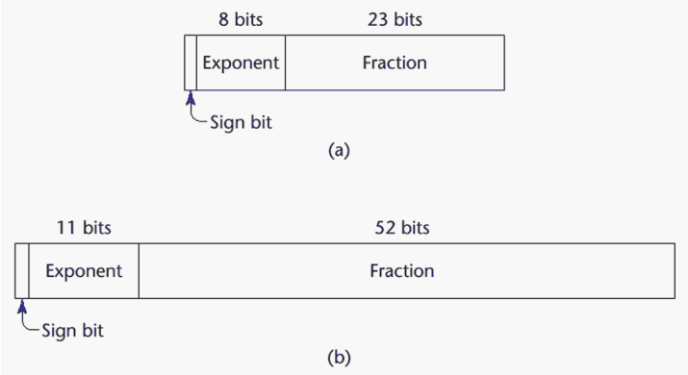
- Practicamente todos los lenguajes de programación proveen un conjunto de tipos de datos primitivos
- Los tipos de datos primitivos son aquellos que no pueden definirse en terminos de otros tipos de datos
- Algunos tipos de datos son simplemente reflejos del hardware
- Otros requieren un poco de soporte no-hardware

Tipos de datos primitivos: *Integer*

- Caso siempre es un reflejo exacto del hardware de forma tal que el mapeo es trivial
- Pueden existir como ocho tipos de *integer* en un lenguaje
- Los tamaños de los enteros con signo en Java: `byte, short, int, long`

Tipos de datos primitivos: Floating Point

- Modela números reales, pero solo como aproximaciones
- Los lenguajes para uso científico tienen al menos dos tipos de floating-point (`float` y `double`);
- Usualmente son representaciones exactas del hardware pero no siempre
- IEEE Floating-Point Standard 754



Tipos de datos primitivos: Decimal

- Para aplicaciones de negocio (money)
  - Esencial para COBOL
  - C# ofrece un tipo decimal
- Guardan un número fijo de dígitos decimales
- *Ventaja*: precisión
- *Desventaja*: rango limitado, desperdicio de memoria

Tipos de datos primitivos: Boolean

- El más simple de todos
- Rango de valores: dos elementos, uno para “true” y uno para “false”
- Puede implementarse como bits, pero usualmente como bytes
  - Ventaja: legibilidad

Tipos de datos primitivos: Character

- Guardado como una codificación numérica
- El código más común: ASCII
- Una alternativa de 16-bit : Unicode
  - incluye caracteres de la mayoría de los lenguajes naturales
  - Originalmente utilizado en Java
  - C# y JavaScript también soportan Unicode

Tipos de cadenas de caracteres
Character String Types

- Los valores son secuencias de caracteres
- Cuestiones de diseño:
  - Es un tipo primitivo o es solo un tipo especial de array?
  - El tamaño de las cadenas debería ser estático o dinámico?

Operaciones sobre cadenas de caracteres

- Operaciones típicas:
  - Asignación y copiado
  - Comparación (=, >, etc.)
  - Concatenación
  - Referencia a subcadena
  - Busqueda de patrones

Tipo de cadenas de caracteres en algunos lenguajes

- C y C++
  - No primitivo
  - Usar **`char`** arrays y librería de funciones para proveer las operaciones
- SNOBOL4 (string manipulation language)
  - Primitivo
  - Muchas operaciones, incluyendo un mecanismo elaborado de pattern matching
- Java
  - Primitivo vía la clase `String`

Opciones de tamaño de cadena de caracteres

- *Estático*: COBOL, Java's `String` class
- *Tamaño dinámico limitado*: C y C++
  - En los lenguajes basados en C, un caracter especial es utilizado para indicar el fin de una cadena de caracteres en vez de mantener el tamaño
- *Dinámico* sin máximo: SNOBOL4, Perl, JavaScript
- Ada soporta todas las opciones de tamaño

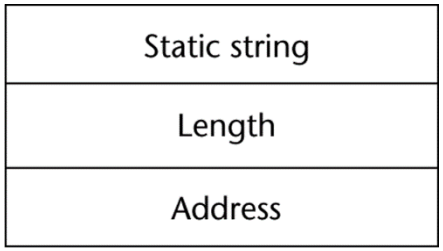
Evaluación de los tipos de cadena de caracteres

- Añade facilidad de escritura
- Cómo un tipo primitivo con tamaño estático, son fáciles de proveerlos –por qué no tenerlos?
- El tamaño dinámico es bueno pero vale su precio?

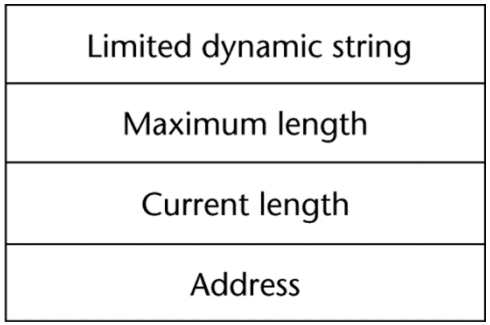
Implementación de cadenas de caracteres

- *Static length*: descriptor en tiempo de compilación
- *Limited dynamic length*: puede requerir un descriptor en tiempo de ejecución para el tamaño (no en C y C++)
- *Dynamic length:* necesita un descriptor en tiempo de ejecución, asignación y desasignación es el mayor problema de implementación

Descriptores en tiempos de compilación y ejecución



Descriptores en tiempo de compilación para strings estáticos



Descriptores en tiempo de ejecución para strings dinámicos limitados

Tipos ordinales definidos por los usuarios
User-Defined Ordinal Types

- Un tipo ordinal (ordinal type) es uno para el cual el rango de los valores posibles puede ser fácilmente asociado con el conjunto de enteros positivos
- Ejemplos de tipos primitivos ordinales en Java
  - `integer`
  - `char`
  - `boolean`

Tipo Enumeración

- Todos los valores posibles, que son constantes con nombres, se proveen en la definición
- Ejemplo en C#

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

- Cuestiones de diseño
  - Se permite que una constante enumeración aparezca en más de una definición de tipo, y si es así como se verifica la ocurrencia de tal constante?
  - Son los valores enumeración convertidos (coerced) a *integer*?
  - Se puede convertir de otro tipo a un tipo enumeración?

Evaluación de los tipos enumeración

- Ayuda a la legibilidad, ejemplo: no se necesita codificar un color como un número
- Ayuda a la confiabilidad, ej. el compilador puede verificar:
  - operaciones (no permitiendo sumar colores)
  - ninguna variable enumeración puede tener asignado un valor fuera de su rango definido
  - En Ada, C#, y Java se provee un mejor soporte para la enumeración que en C++ puesto que las variables enumeración no se convierten en tipos enteros

Subrange Types (Subrangos)

- Una secuencia contigua ordenada de un tipo ordinal
- Ejemplo: 12..18 es un subrango de un tipo entero
- Ada:

```
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;
```

Evaluación de subrangos

- Ayudad a la legibilidad
  - Hacen claro para los lectores que las variables de un subrango pueden guardar cierto rango de valores
- Confiabilidad
  - Asigna un valor a una variable subrango que está fuera del rango especificado es detectado como un error

Implementación de tipos ordinales definidos por los usuarios

- Los tipos enumeración son implementados como enteros
- Los tipos subrango son implementados como los tipos padres con código insertado (por el compilador) para restringir las asignaciones a los subrangos de las variables

Tipo Array

- Un array es un tipo agregado de elmentos de datos homogeneos en el cual un elemento individual se identifica por medio de su posición relativa al primer elemento

Cuestiones de diseño de los array

- Qué tipos son legales para los subíndices?
- Se chequean los rangos de los subindices?
- Cuando se asocian los subindices?
- Cuando ocurre la asignación de memoria?
- Cuál es el número máximo de subíndices?
- Pueden los arrays ser inicializados?
- Se permiten los slices?

Indexación de Arrays

- *Indexación - Indexing* (subscripting) es un mapeo de los índices a los elementos

```
array_name (index_value_list) →  an element
```

- Sintaxis de los índices
  - FORTRAN, PL/I, Ada usa paréntesis
    - Ada explícitamente utiliza paréntesis para mostrar uniformidad entre referencias de array y llamadas a funciones porque ambos son mapeos
  - La mayor parte de los lenguajes usan corchetes

Arrays Index (Subscript) Types

- FORTRAN, C: solo entero
- Pascal: cualquier tipo ordinal (integer, Boolean, char, enumeration)
- Ada: entero o enumeración (incluye Boolean y char)
- Java: sólo tipo entero
- C, C++, Perl, y Fortran no especifican un verificación del rango
- Java, ML, C# especifican un chequeo del rango

Asociación de subíndices y categorías de arrays

- *Static*: rangos de subíndices son asociados estáticamente y la asignación del almacenamiento es estático (antes del tiempo de ejecución)
  - Ventaja: eficiencia (no existe necesidad de una asignación dinámica)
- *Fixed stack-dynamic*: los rangos de los subindices son determinados de manera estática pero la ubicación se hace en tiempo de declaración
  - Ventaja: eficiencia del espacio

Asociación de subíndices y categorías de arrays

- *Stack-dynamic*: los rangos de los subíndices son asociados de manera dinámica y la ubicación de almacenamiento de manera dinámica (en run-time)
  - Ventaja: flexibilidad (la longitud del array no necesita conocerse hasta que se precise utilizarlo)
- *Fixed heap-dynamic*: similar a fixed stack-dynamic: la asociación del almacenamiento es dinámico pero fijo luego de la ubicación (i.e., la asociación se realiza cuando se solicita y el almacenamiento es realizado en el heap no en el stack)

Asociación de subíndices y categorías de arrays

- Heap-dynamic: la asociación de los rangos de los subíndices y la asignación del almacenamiento es dinámica y puede cambiar una gran cantidad de veces
  - Ventaja: flexibilidad (los arrays pueden crecer o reducirse durante la ejecución del programa)

Asociación de subíndices y categorías de arrays

- Los arrays en C y C++ que incluyen el modificador `static` son estaticos
- Los arrays en C y C++ sin el modificador `static` son *fixed stack-dynamic*
- Los arrays en Ada pueden ser stack-dynamic
- C y C++ proveen fixed heap-dynamic arrays
- C# incluye un segundo tipo de array `ArrayList` que provee fixed heap-dynamic
- Perl y JavaScript soportan arrays heap-dynamic

Inicialización de Arrays

- Algunos lenguajes permiten la inicialización en el momento de la asignación del almacenamiento
  - C, C++, Java, C#

```
int list [] = {4, 5, 7, 83}
```

  - Cadenas de caracteres en C y C++

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

  - Arrays de strings en C y C++

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

  - Inicialización en Java de objetos string

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

Operaciones sobre Arrays

- APL provee el conjunto de operaciones sobre arrays más poderoso para vectores y matrices así como operadores unarios (por ejemplo, para dar vuelta elementos de columnas)
- Ada permite asignación sobre arrays y concatenar
- Fortran provee operaciones elementales entre pares de elementos
  - Por ejemplo, el operador + entre dos arrays resulta en la suma de los pares de elementos entre los dos arrays

Arrays rectangulares y Jagged

- Un array rectangular es un array multidimensional en el cual todas las filas y columnas tienen en mismo número de elementos
- Una *jagged matrix* tiene filas con un número variable de elementos
  - Es posible cuando los arrays multidimensionales son arrays de arrays

Slices

- Un slice es una subestructura de un array; nada más que un mecanismo de referenciamiento
- Los Slices son útiles solamente en lenguajes que tienen operaciones sobre arrays

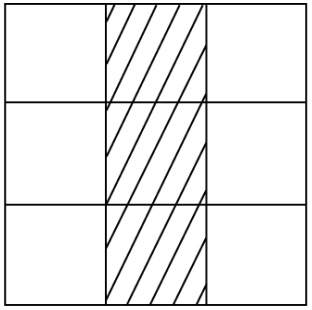
Slices (Ejemplos)

- Fortran 95

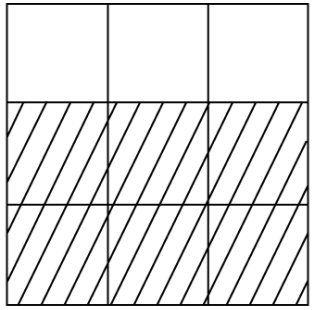
```
Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3) :: Cube
```

`Vector (3:6)` es un array de cuatro elemntos

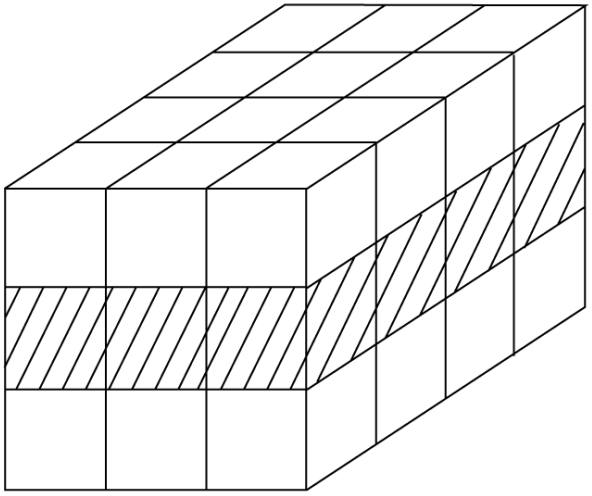
Slices en Fortran 95



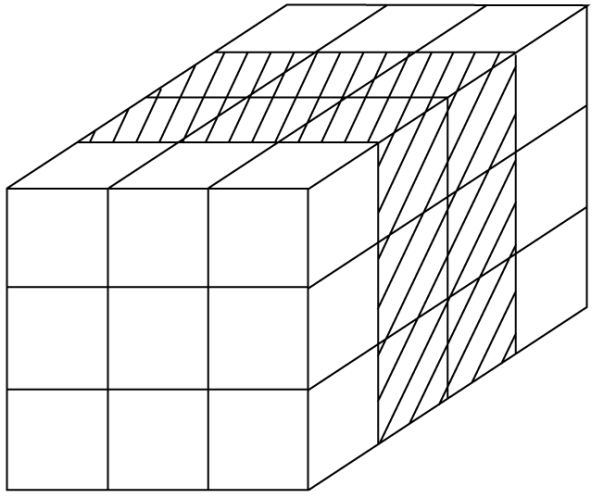
MAT (1:3, 2)



MAT (2:3, 1:3)



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



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

Implementación de Arrays

- Las funciones de acceso mapea las expresiones de los subíndices a una dirección en el array
- La función de acceso para arrays unidimensionales :

  address(list[k]) = address (list[lower_bound])

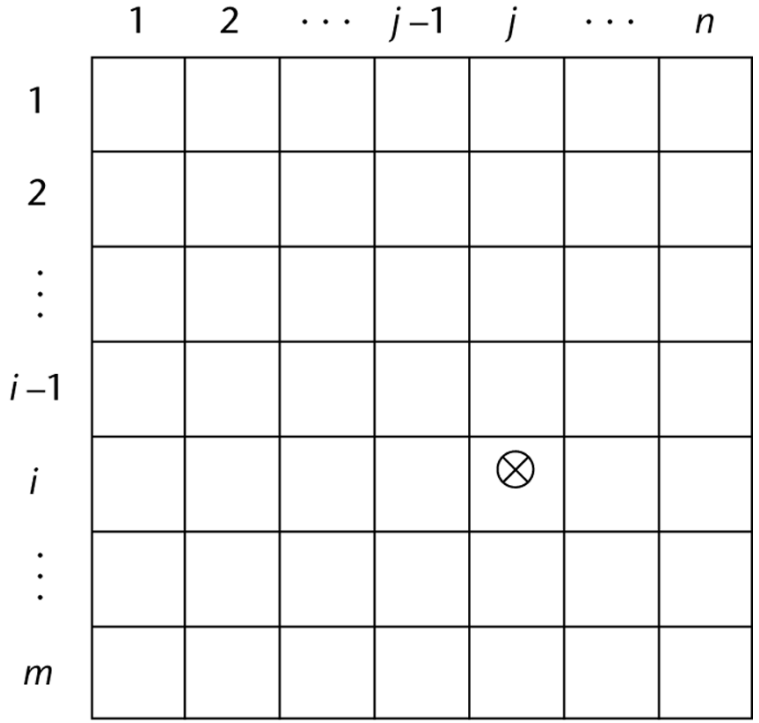
  + ((k-lower_bound) * element_size)

Acceso a arrays multidimensionales

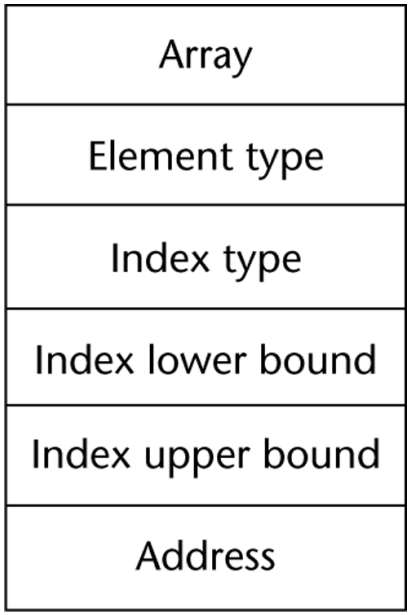
- Dos formas comunes:
  - Row major order (por filas) – usado en la mayoría de los lenguajes
  - column major order (por columnas) – usado en Fortran

Ubicando un elemento en un array multidimensional

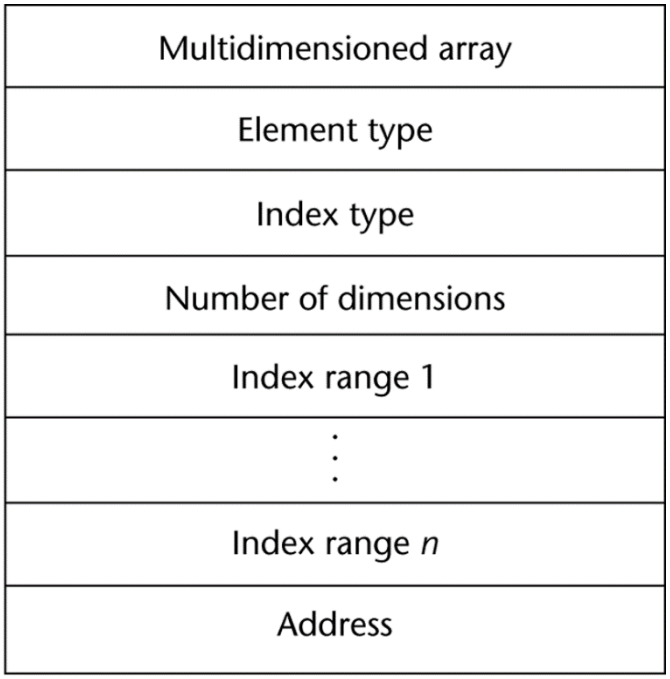
Forma general:
Location (a[I,j]) = address of a [row_lb,col_lb] + (((I – row_lb) * n) + (j – col_lb)) * element_size



Descriptores en Compile-Time



Array unidimensional



Array Multi-dimensional

Arrays associativos

- Un array asociativo es una colección no ordenada de elementos de datos que pueden ser indexados por un número igual de claves (*keys*)
  - Las claves definidas por el usuario deben ser guardadas
- Cuestiones de diseño: cual es la forma de las referencias a los elementos

Associative Arrays en Perl

- Los nombres empiezan con `%;` los literales se delimitan con paréntesis

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

- La subindización se hace usando llaves y claves

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

  - Los elementos pueden eliminarce con `delete`

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

Registros (*Record Types*)

- Un *record* es una tipo de dato posiblemente agregado de elementos de datos en los cuales los elementos individuales son identificados con nombres
- Cuestiones de diseño:
  - Cuál es la forma sintáctica de las referencias de los campos?
  - Son permitidas referencias elipticas

Definición de registros

- COBOL utilizas número de niveles para mostrar registros anidados; otros utilizan definiciones recursivas
- Referencias a los campos de los registros

  1. COBOL

  field_name OF record_name_1 OF ... OF record_name_n

  2. Otros (notación punto - *dot notation*)

  record_name_1.record_name_2. ... record_name_n.field_name

Definición de registros en COBOL

- COBOL usa número de niveles para mostrar los registros anidados; otros utilizan una definición recursiva

```
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.
```

Definición de registros en Ada

- Las estructuras de los registros son indicadas de una forma ortogonal

```
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;
```

Referencias a registros

- La mayoria de los lenguajes usan notación punto

  `Emp_Rec.Name`

- Las referencias recursivas (Elliptical references) haciendo de las referencias no ambiguas, por ejemplo en COBOL

  `FIRST, FIRST OF EMP-NAME`, y `FIRST` of EMP-REC son referencias elipticas al nombre del empleado

Operaciones sobre registros

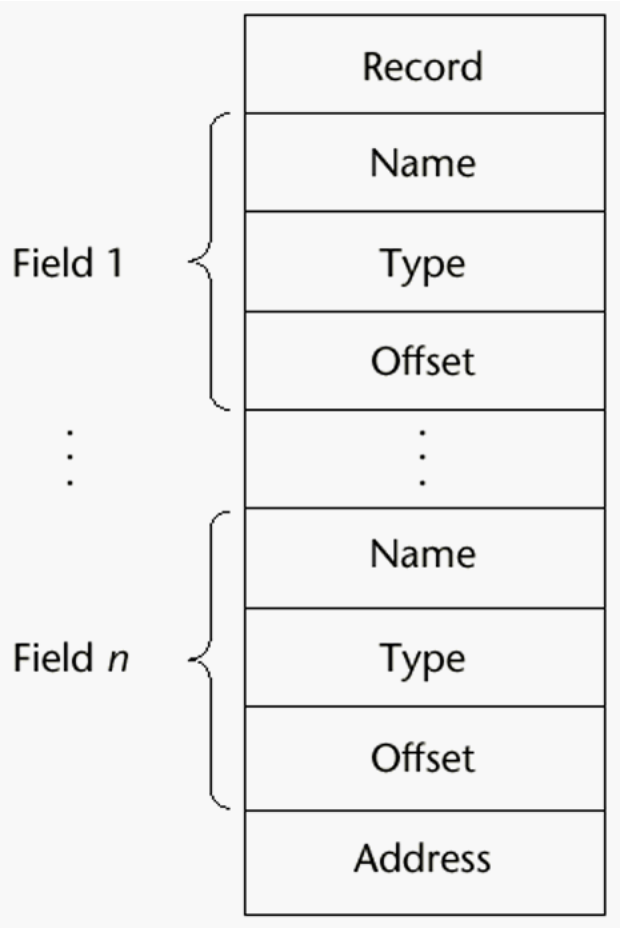
- Las asignaciones son comunes si los tipos son idénticos
- Ada permite comparar registros
- Los registros de Ada pueden ser inicializados con literales agregados
- COBOL provee `MOVE CORRESPONDING`
  - Copia un campo del registro origen al campo correspondiente en el registro destino

Evaluación y comparación con relación a los arrays

- Diseño directo y seguro
- Los registros son utilizados cuando la colección de datos es hetereogenea
- El acceso a los elementos de los arrays es mucho más lento que a los campos de los registros puesto que los subindices son dinámicos (los nombres de campos estáticos)

Implementación del tipo registro

La dirección del Offset relativa al comienzo de los registros es asociado con cada campo



Tipo union

- Una *union* es un tipo cuyas variables se pueden utilizar para guardar diferentes valores de diferentes tipos durante la ejecución
- Cuestiones de diseño
  - Se requiere chequeo de tipo?
  - Deberían ser las uniones incorporadas en los registros?

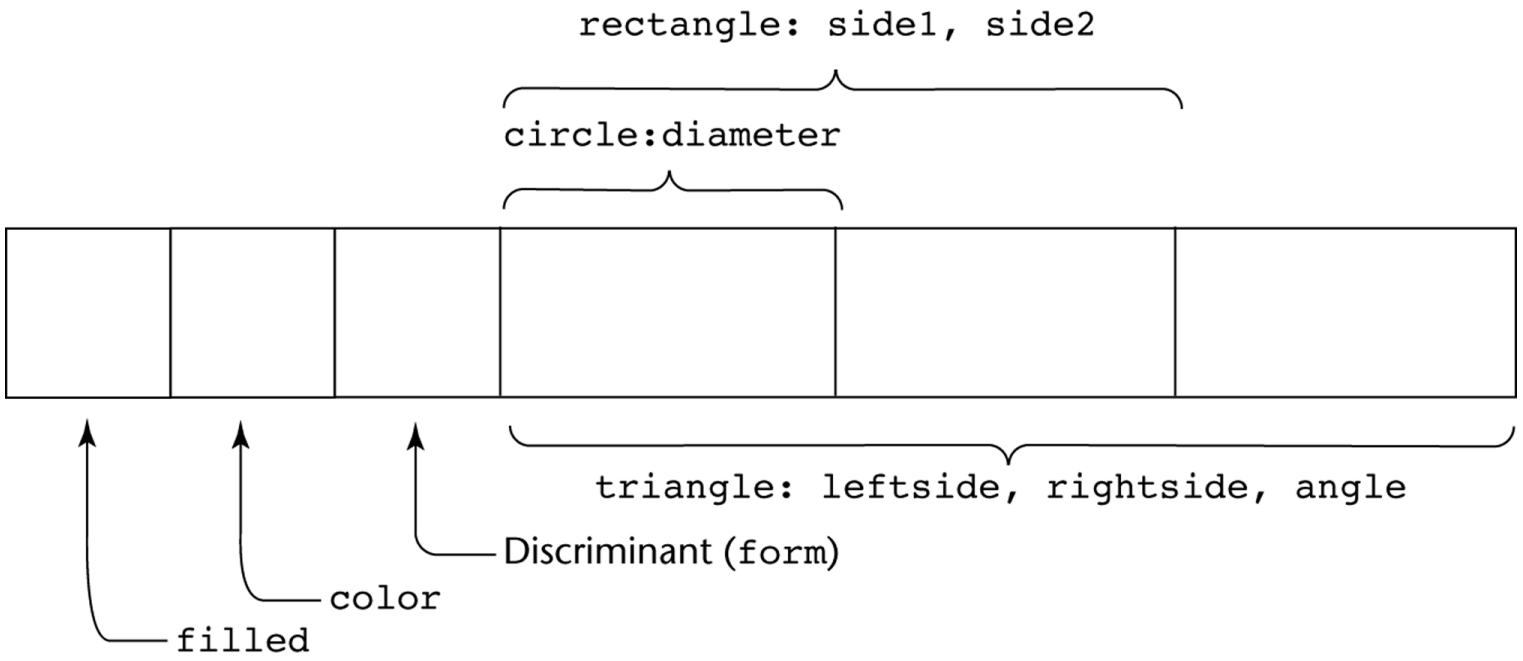
Discriminated vs. Free Unions
Uniones discriminadas/libres

- Fortran, C, y C++ proveen constructores de union en donde no existe soporte del lenguaje para el chequeo de tipo; las uniones en estos lenguajes se llaman libres
- La verificación de tipo de las uniones requiere que cada unión utilice un indicador de tipo llamado su discriminante
  - Soportado por Ada

Tipo Union en Ada

```
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;
end record;
```

Tipo union de Ada



Una unión discriminante para tres variables de formas

Evaluación de las uniones

- Construcciones potencialmente inseguras
  - No permiten chequeo de tipo
- Java y C# no soportan uniones
  - Reflejo de la preocupación creciente por la seguridad en los lenguajes de programación

Punteros y tipos referencia

- Una variable de tipo puntero tiene un rango de valores que consiste de direcciones de memoria y un valor especial, *nil*
- Provee el poder de direccionamiento indirecto
- Provee una forma de manejar memoria dinámica
- Un puntero puede ser utilizado para acceder a una ubicación en el área donde se realiza el almacenamiento dinámico (*heap*)

Cuestiones de diseño para los punteros

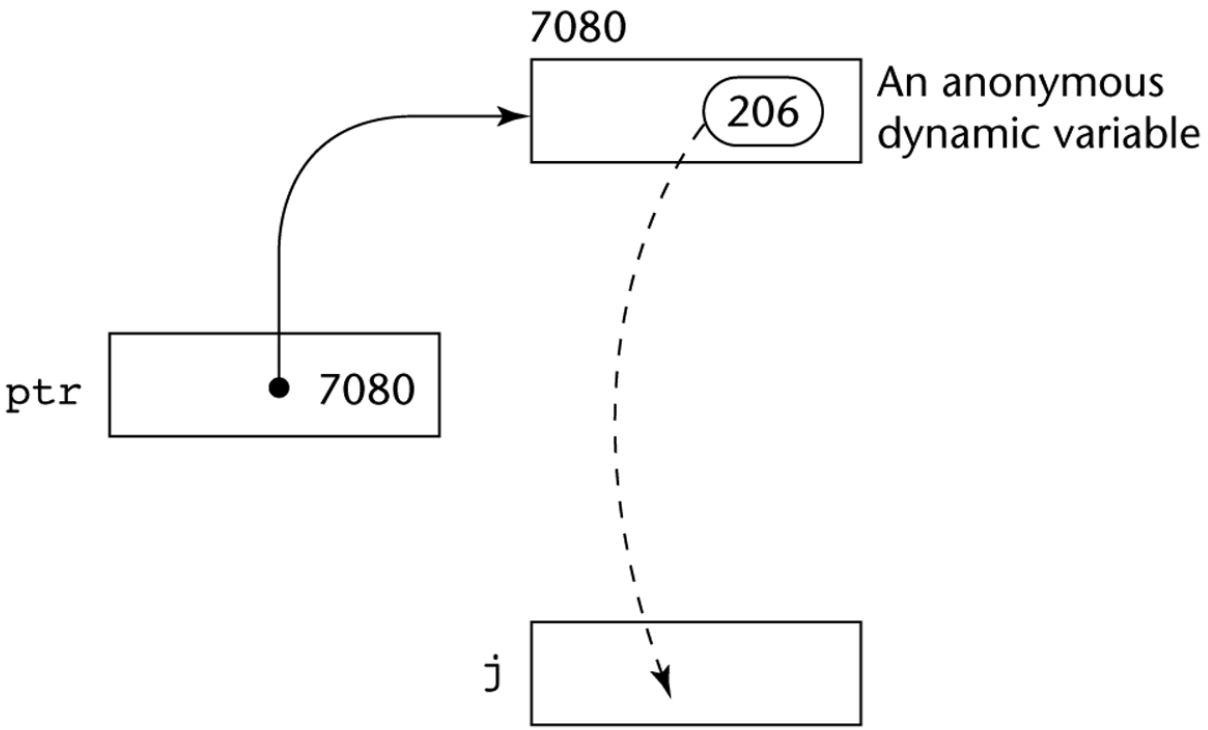
- Cuál es el alcance y el tiempo de vida de una variable puntero?
- Cuál es el tiempo de vida de una variable heap-dynamic variable?
- Son los punteros restringidos a aquellos tipos de valores que pueden apuntar?
- Son los punteros utilizados para la administración del almacenamiento dinámico, el direccionamiento indirecto o ambos?
- Deben los lenguajes soportar punteros, tipos de referencia o ambos?

Operaciones sobre punteros

- Asignación y desreferenciamiento
- La asignación permite setear la variable a una dirección util
- El dereferenciamiento permite obtener el valor
  - El dereferencing puede ser explícito o implícito
  - C++ utiliza una forma explícita *

```
j = *ptr
```

Asignación de punteros



j = *ptr

Problemas con punteros

- Dangling pointers (peligroso)
  - Un puntero apuna a una variable heap-dynamic que se desasigno
- Variable heap-dynamic perdida
  - Una variable heap-dynamic variable que ya no es accedida por el programa (*garbage*)
    - Puntero `p1` es seteado para apuntar a una variable heap-dynamic nueva
    - puntero `p1` es seteado para apuntar a otro lado

Punteros enAda

- Algunos punteros suspendidos (dangling) son evitados puesto que los objetos dinamicos son automáticamente desasignados al final del alcance del puntero
- El problema de la variable perdida en el heap es eliminado

Punteros en C y C++

- Extremadamente flexibles pero deben ser utilizados con cuidado
- Pueden apuntar a cualquier variable
- Usado para el manejo de la memoria dinámica y direccionamiento
- Es posible la aritmética de punteros
- Operadores de dereferenciamiento explícito y dirección de
- El tipo puede no ser fijo (**`void *`**)
- `void *` puede apuntar a cualquier tipo(no puede desreferenciarse)

Aritmética de punteros en C y C++

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

`*(p+5)` es equivalente a `stuff[5]` y `p[5]`

`*(p+i)` equivalente a `stuff[i]` y `p[i]`

Tipos referencia

- C++ incluye un tipo especial de puntero llamado *reference type* que se usa para los parámetros formales
  - Ventaja: pass-by-reference y pass-by-value

Evaluación de punteros

- Problemas punteros y objetos perdidos, manejo del heap
- Punteros son como `goto` pueden acceder a cualquier celda
- Punteros o referencias son necesarios para manejar las estructuras de datos dinámicas

Representaciones

- Valores únicos
- Segment y offset (intel)

Resumen

- Los tipos de datos de un lenguaje son una parte importante que determina el estilo y la utilidad del lenguaje
- Los tipos primitivos de datos de la mayor parte de los lenguajes imperativos incluyen: numéricos, caracter y booleanos
- Las enumeraciones definidas por los usuarios y los subrangos agregan legibilidad y confiabilidad a los programas
- Los arrays y los registros se incluyen en la mayoría de los lenguajes
- Los punteros son utilizados para agregar flexibilidad al area de almacenamiento dinámico

Proxima clase

Dangling Pointer Problem

- *Tombstone*: una celda extra en el heap que es un puntero a la variable heap-dynamic
  - El puntero solo apunta a los tombstones
  - Cuando de desasigna una variable heap-dynamic variable, el tombstone se mantiene pero se setea a nil
  - Costoso en tiempo y espacio

. *Locks-and-keys*: valores punteros representados como pares (key, address)

  - Las variables heap-dynamic variables son representadas como variables más una celda para el bloqueo (entero)
  - Cuando se asigna una variable heap-dynamic, el valor del lock se crea y se ubica en la celda de bloqueo y clave del puntero