Lenguajes de Programación I

Tipos de Datos Compuestos

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Registros

. . . o Estructuras (*Structures*)

- Agrupan valores de tipos heterogéneos – $n-$upla matemática.
- Se almacenan y manipulan como un todo.
- Cada componente se denomina **campo** (*field*).
- La estructura define un nuevo tipo – pueden anidarse.
  - Modelo de valor – anidar es equivalente a incluir.
  - Modelo de referencia – anidar es contener una referencia.

Declarando Registros

Orden posicional de los campos

- En C

```
struct elemento {
  int    numero_atomico;
  char   nombre[2];
  double peso_atomico;
  int    es_metalico;
};
```

- En Pascal

```
type elemento = record
  numero_atomico : integer;
  nombre         : array[1..2] of char;
  peso_atomico   : real;
  es_metalico    : boolean
end;
```

Accediendo a los datos organizados en registros

Funcionan como el alcance

- Operador de acceso a los campos – tradicionalmente es el punto.
- En C

```
struct elemento carbono;
carbono.numero_atomico = 12;
if (carbono.es_metalico) { ... }
```

- En Pascal

```
var cobre : elemento;
cobre.nombre      := 'Cu';
cobre.es_metalico := true;
```

- En Fortran se escribe `carbono%nombre`.

Almacenamiento de Registros

En lenguajes compilados

- Campos se almacenan en ubicaciones contiguas de memoria.
- Compilador calcula los *desplazamientos* desde la base de almacenamiento hasta cada campo.
- Optimizar para velocidad – lo usual.
  - Campos alineados con límites de palabra en hardware.
  - Espacios vacíos entre los campos.
- Optimizar para espacio – opcional en Pascal.
  - **Registros empaquetados** (*packed*) – no hay espacio entre los campos.
  - Necesita más instrucciones para acceso.
- Lo mejor de ambos mundos – costoso de implantar.
  - Reordenar los campos.
  - Maximizar velocidad de acceso, minimizar desperdicio.

Velocidad vs. Espacio

Depende de restricciones de la arquitectura

- Cada arquitectura tiene sus propias limitaciones.
- **Representación**
  - ¿Cuánto espacio ocupa un valor del tipo de datos?
  - Ligada al tamaño de los registros e instrucciones disponibles.
- **Alineación**
  - ¿Dónde es *mejor* almacenar el valor del tipo de datos?
  - Ligada a los modos de direccionamiento del procesador.
  - Usualmente múltiplos pares (de 2, de 4 o de 8 bytes).
- La presencia de arreglos suele afectar ambas restricciones.

Siempre son arbitrarias y respetarlas mejora el desempeño.

¿Cómo organizamos este tipo de datos?

Con mucho cuidado...

```
struct meta {
  int    foo;
  char   bar[2];
  int    baz;
  double qux;
  bool   meh;
};
```

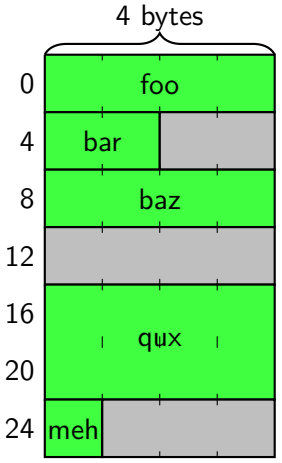
Tipo	Representación	Alineación
`bool`	1 byte	2 bytes
`char`	1 byte	2 bytes
`int`	4 bytes	4 bytes
`float`	8 bytes	8 bytes

- Arreglos de `char` se *alinean* al principio y se *almacenan* contiguos – comprenderán la razón en unos minutos...
- Restricciones similares a una arquitectura de 32 bits CISC.

Organización para eficiencia de acceso

Respetar alineación aún a costa de espacio desperdiciado

```
struct meta {
  int    foo;
  char   bar[2];
  int    baz;
  double qux;
  bool   meh;
};
```

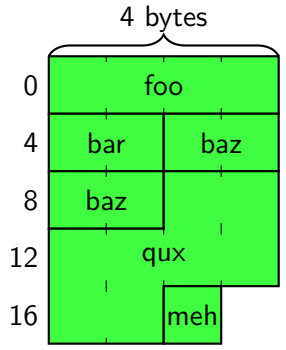


- Dispuestos en orden de aparición.
- Primer campo **siempre** alineado según las reglas.
- Resto alineados de forma óptima según las reglas.
- Ocupación: 28 bytes.
- Desperdicio: 9 bytes.

Organización para eficiencia de espacio

Ignorar la alineación disponiendo un campo detrás de otro

```
struct meta {
  int    foo;
  char   bar[2];
  int    baz;
  double qux;
  bool   meh;
};
```

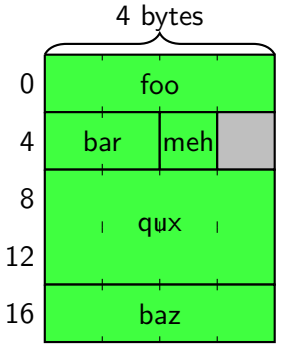


- Dispuestos en orden de aparición.
- Primer campo **siempre** alineado según las reglas.
- No se dejan espacios vacíos entre campos.
- Ocupación: 28 bytes.
- Desperdicio: 0 bytes.

Dispo-zen-ción balanceada

Reordenar los campos según sea necesario

```
struct meta {
  int    foo;
  char   bar[2];
  int    baz;
  double qux;
  bool   meh;
};
```



- Maximizar velocidad, minimizar espacio.
- Reordenar es difícil – es NP-completo.
- Ocupación: 20 bytes.
- Desperdicio: 1 byte.

Registros Variantes

Espacio común con múltiples interpretaciones

- Colecciones alternativas de campos que comparten espacio.
- Sólo *una* de las alternativas es válida en cualquier momento.
- Se originan en los `equivalence` de Fortran

```
integer i
real r
logical b
equivalence (i, r, b)
```

- Concebidas originalmente para ahorrar espacio – aún son útiles en programación de bajo nivel.

Registros Variantes

`union` en C

```
struct elemento {
  char   nombre[2];
  int    numero_atomico;
  double peso_atomico;
  int    es_metalico;
  int    es_natural;
  union {
    struct {
      char *donde;
      double prevalencia;
    } natural;
    double tiempo_de_vida;
  } extra;
} cobre;
```

¿Cómo almacenarlos?

Con mucho más cuidado...

- Cada alternativa se organiza como si fuese un registro independiente.
  - Aplicar las restricciones de la plataforma.
  - Optimizar para velocidad o espacio según convenga.
- Registro variante ocupará tanto como la alternativa que ocupe más.
- De estar anidado en un registro, es necesario alinear el primer campo de cada variante como si fuese el siguiente campo en el registro.

¿Estamos usando la interpretación válida?

Depende de la definición del lenguaje

- ¿Cómo *garantizar* que se usa la alternativa "actual"?
- Dejarlo a riesgo del programador – caso extremo de *aliasing*.
- Forzarlo en el lenguaje de manera explícita.
  - Exigir un *campo discriminante* – tipo discreto.
  - Obligatorio asignar *todos* los campos al cambiar el discriminante.
  - Verificable estáticamente.
- Forzarlo en el lenguaje de manera implícita.
  - Campo escondido que establece la alternativa vigente – tomarlo en cuenta para la reserva de espacio.
  - Emitir código para mantener el campo vigente después de cada *asignación* a campos variantes.
  - Emitir código para verificar el campo vigente antes de cada *lectura* de campos variantes.

Arreglos

El más común e importante de los tipos compuestos

- Asociación entre un *tipo índice* y un *tipo componente* o *almacenado*.
- En la mayoría de los lenguajes el tipo índice es entero, pero muchos lenguajes permiten cualquier tipo discreto.
- Se hace referencia a un elemento dentro de un arreglo con un subíndice delimitado
  - En Fortran y Ada se usa `A(42)`.
  - En Pascal y C se usa `A[42]`.

Declaración de Arreglos

- Agregando notación subíndice a una declaración escalar.
  - C/C++ con la base en cero

    ```
    char letras[26];
    ```

  - En Fortran con la base en uno

    ```
    character dimension (1:26) :: letras
    ```

- Con un constructor especial.
  - Como en Pascal

    ```
    var letras : array ['a'..'z'] of char
    ```

  - Como en Ada

    ```
    letras : array (character range 'a'..'z')
             of character
    ```

- Arreglos multidimensionales.

Operaciones sobre Arreglos

El todo es más que la suma de sus partes

- Acceso directo por subíndice es el habitual.
- Algunos lenguajes permiten sobrecargar el operador de indexado (caso `operator[]` en C++, `indexer` en C#).
- Secciones de Arreglo (*slices*) como en Fortran, Perl y R.
- Fortran90 permite
  - Comparar arreglos como un todo.
  - Aritmética sobre arreglos para obtener un arreglo resultado – facilitar operaciones "vectoriales" o "matriciales".
  - Búsqueda, transposición, permutación de subíndices.

Dimensiones, Límites y Almacenamiento

La forma de vida de los arreglos

- **Forma** de un arreglo puede determinarse estática o dinámicamente.
- **Vida Global, Forma Estática** – se almacena estáticamente.
- **Vida Local, Forma Estática** – se almacena en la pila.
- **Vida Local, Forma Estática al elaborar** – se almacena en la pila con una indirección (*parte fija* vs. *parte variable* usando *dope vectors*).
- **Vida Arbitraria, Forma Estática al elaborar** – (Java, C#) se almacenan en el *heap* y su tamaño nunca cambia.
- **Vida Arbitraria, Forma Dinámica** – se almacena en el *heap* y su tamaño cambiante obliga a reubicación.

Disposición en Memoria

Proyección a una dimensión

- Se almacenan en posiciones contiguas de memoria – manteniendo la alineación de la plataforma.
- Para arreglos multidimensionales
  - **Orden por filas** (*row major order*) – las dimensiones se hacen variar de última a primera.
  - **Orden por columnas** (*column major order*) – las dimensiones se hacen variar de primera a última.
  - **Disposición con Apuntadores** (*row pointer layour*) – filas contiguas en memoria, con apuntadores al primer elemento
    - Permiten filas de longitud variable – arreglos esparcidos o de cadenas.
    - Requiere código extra y con indirecciones.
- Fortran es el único que usa *column major*.
- C, C++ y C# permiten *row-pointer*.
- Java sólo tiene *row-pointer*.

Calculando las direcciones

Cálculo complicado pero directo

- Supongamos una declaración de la forma
  `foo : array [`L_1`..`U_1`] of array [`L_2`..`U_2`] of array [`L_3`..`U_3`] of e_type;`
- El tamaño de cada dimensión puede calcularse estáticamente

$$S_3 = sizeof(\textbf{e\_type})$$
$$S_2 = (U_3 - L_3 + 1) \times S_3$$
$$S_1 = (U_2 - L_2 + 1) \times S_2$$

- ¿Cuál es la dirección de `foo[i,j,k]`?

$$addressof(foo) + (i - L_1) \times S_1 + (j - L_2) \times S_2 + (k - L_3) \times S_3$$

Calculando las direcciones

Reduciendo la complejidad del cómputo

$$addressof(foo) + (i - L_1) \times S_1 + (j - L_2) \times S_2 + (k - L_3) \times S_3$$

- Muy costoso calcularlo – 6 sumas y 3 multiplicaciones.
- Pero S_p y L_p son *constantes* conocidas a tiempo de compilación – podemos expandir la expresión...

$$addressof(foo) + i \times S_1 - L_1 \times S_1 + j \times S_2 - L_2 \times S_2 + k \times S_3 - L_3 \times S_3$$

...y reordenarla un poco

$$addressof(foo) - (L_1 \times S_1 + L_2 \times S_2 + L_3 \times S_3) + (i \times S_1 + j \times S_2 + k \times S_3)$$

¡Los dos primeros términos son *constantes* a tiempo de compilación!
Ahora son 3 sumas y 3 multiplicaciones en el peor caso.

Cadenas

- Arreglo de Caracteres vs. Entidades Especiales.
- Cadenas literales entre comillas simples o dobles.
- Caracteres literales vs. cadenas literales (como en C).
- Secuencias de escape para caracteres especiales.
- Longitud variable vs. longitud constante.
- Operaciones disponibles en el lenguaje vs. librerías.

Bibliografía

- [*Scott*]
  - Secciones 7.3 a 7.5
  - Ejercicios y Exploraciones