Programación : repaso de C (3)

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Agosto 2009

Outline

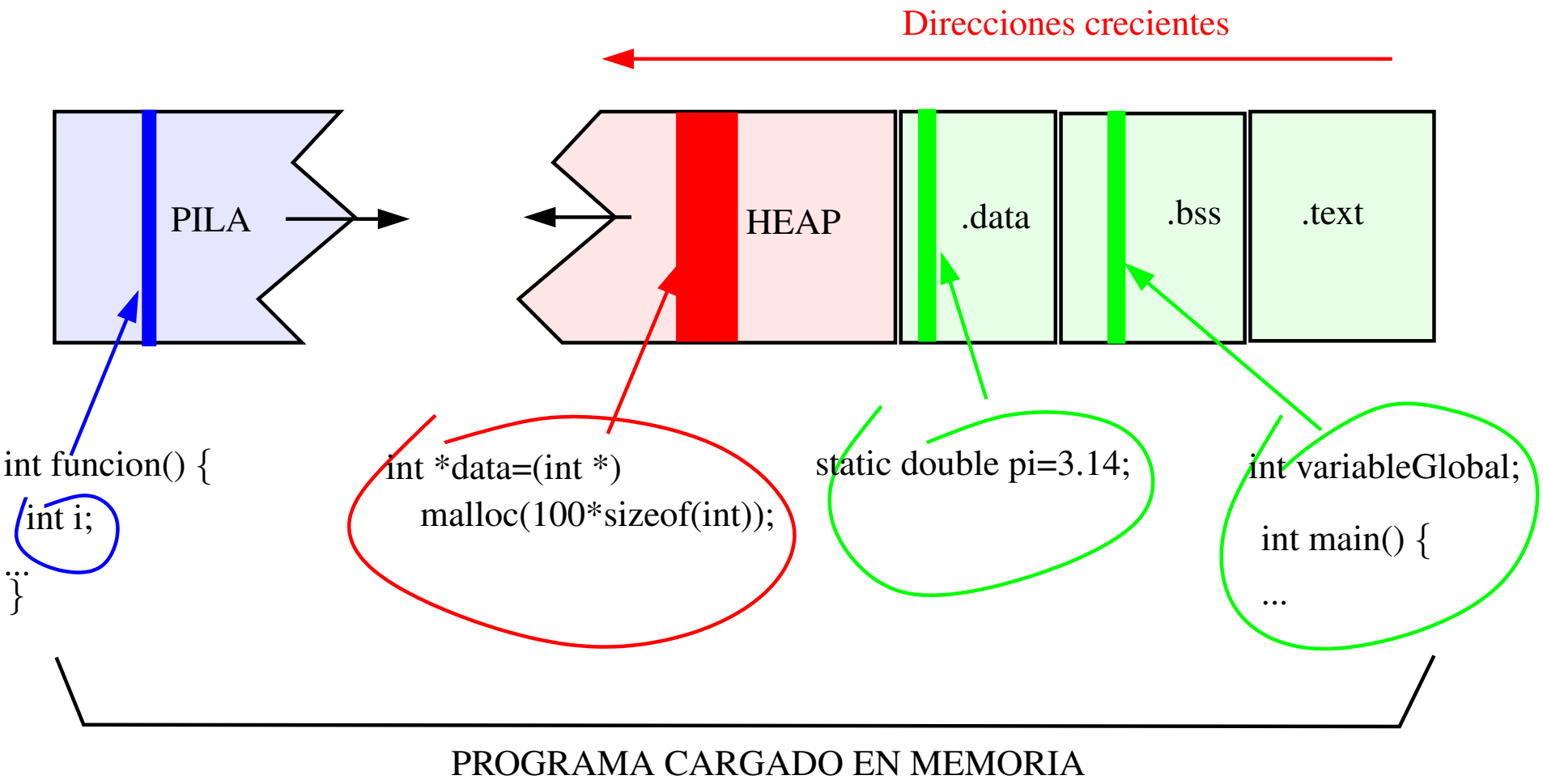
Outline

Memoria

Hay tres tipos de segmentos de memoria para programas informáticos :

- memoria alocada estáticamente (especificado antes de la ejecución),
- memoria sobre pila : para variables dentro de una función,
- memoria sobre *heap* : para las alocaciones dinámicas.

Memoria


Direcciones crecientes
PILA
HEAP
.data
.bss
.text
int funcion() {
int i;
...
}
int *data=(int *)
malloc(100*sizeof(int));
static double pi=3.14;
int variableGlobal;
int main() {
...
PROGRAMA CARGADO EN MEMORIA

const

Esta palabra llave hace del variable que modifica un rvalue : no puede ser modificado por una afectación :

```
const int a = 1;
...
a = 3;
```

No compilará :

```
error: assignment of read-only variable a
```

Pero se puede modificar a través de un apuntador, por ejemplo (las libertades del C...). Esencialmente para señalar optimizaciones posibles al compilador.

static

- variables dentro de funciones con existencia fuera del espacio memoria en la pila; tiene otra significación para variables globales...
- comportamiento por default, *automatic*, es que variables declaradas dentro de un bloque/funcion desaparecen afuera.

register

Este modificador hace que la variable, si es posible, esta guardada en un registro de la CPU, y no en memoria. Es útil si, localmente, esta variable esta usada muchas veces (por ejemplo, un indice de un arreglo en ciclos). Es posible que compiladores eficientes pongan esta palabra llave por si mismos. Por definición, no se puede entonces usar el operador &.

Outline

Estructuras

Se puede combinar elementos de tipos diferentes en un solo objeto, una estructura.

```
struct persona {
        unsigned int edad;
        unsigned int altura;
        char nombre[256];
} tonio, elba;
```

Sus diferentes elementos se acceden por el operador . :

```
        unsigned int ed = elba.edad;
```

Estructuras

Para estos tipos compuestos, no hay operadores aritméticos o de comparación. En particular, si uno quiere checar la igualdad de los datos de una estructura, tiene que proveer la funcion de comparación.

```
if (tonio==elba) {
```

no tiene sentido. En C++ por ejemplo, se podría extender el sentido del operador == (*overload*) al caso de otros objetos que los tipos elementales.

Estructuras : especificando tamaño

Se puede especificar exactamente el numero de bits que queremos para cada elemento :

```
struct persona {
        unsigned int edad : 7;
        unsigned int altura : 25;
        char nombre[256];
} tonio, elba;
```

¿Tamaño ?

Uniones

Una unión es otro caso de tipo compuesto. Es diferente de una estructura puesto no contiene espacio memoria para todos los elementos, sino solo el espacio máximo entre los tipos presentes : se usa para poder ver un espacio memoria según varios tipos diferentes (tipo A o tipo B o tipo C)

Uniones

```
union persona {
        unsigned int edad;
        unsigned int altura;
        char nombre[256];
} tonio, elba;

tonio.edad  = 24;
tonio.altura = 175;
printf("Tonio tiene %d años\n",tonio.edad);
```

¿Qué dice el programa ? Cuál es el tamaño de la unión ? ¿Cuál es el tamaño de la estructura equivalente ?

Enumeraciones

Eso es un tipo que enumera varios valores posibles para un objeto.
Permite tener una forma de semántica :

```
enum deporteEnum {Rugby, Football, Tennis, Golf};
```

Enumeraciones

Hay una limitación importante : C no verifica si los valores puestos a un enum están entre los posibles

```
int main() {
typedef enum deporteEnum {
        Rugby,
        Football,
        Tennis,
        Golf
} deporte;
deporte dep1 = 1,dep2 = 4;
deporte dep3 = Football;
printf("dep1 %d dep2 %d dep3 %d \n",dep1,dep2,dep3)
return 0;
}
```

Definiciones por typedef

La palabra llave *typedef* permite definir nuevos tipos, lo que es útil para los tipos compuestos (uniones, enums, estructuras) : evita repetir la definición de la estructura en cada declaración de variable.

Definiciones por typedef

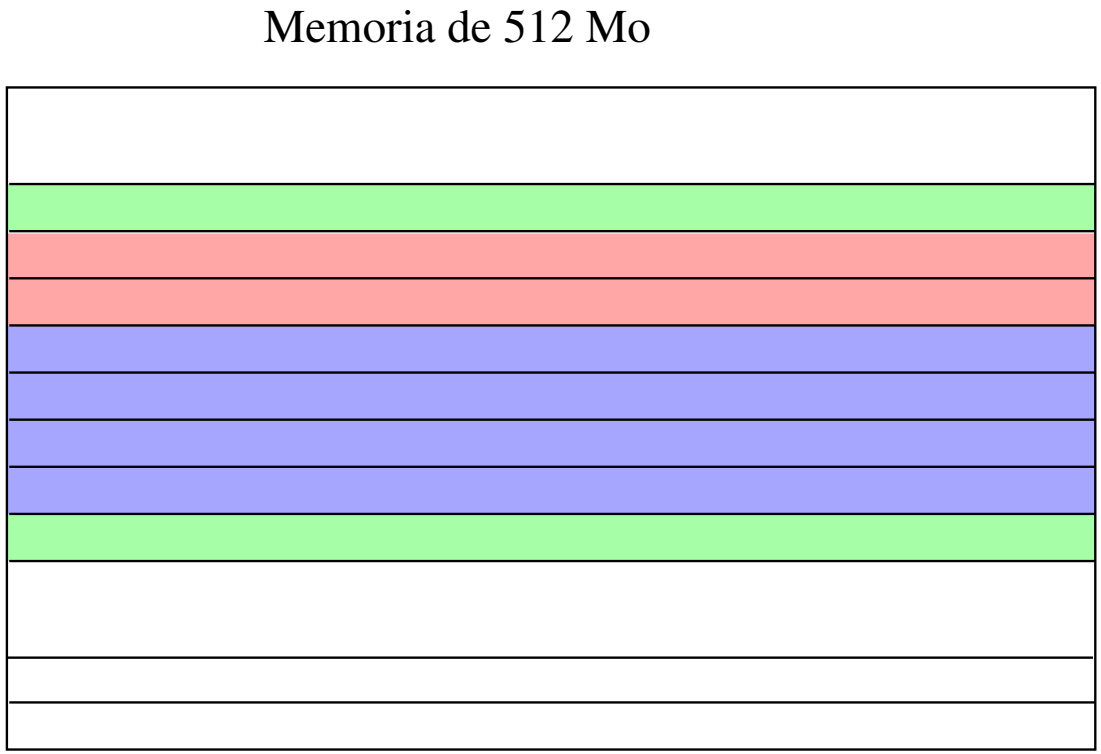
Ejemplo :

```
typedef struct personaStruct {
        unsigned int edad;
        unsigned int altura;
        char nombre[256];
} persona;

persona nery,adolfo,cuahutemoc;
```

Alineamiento de datos

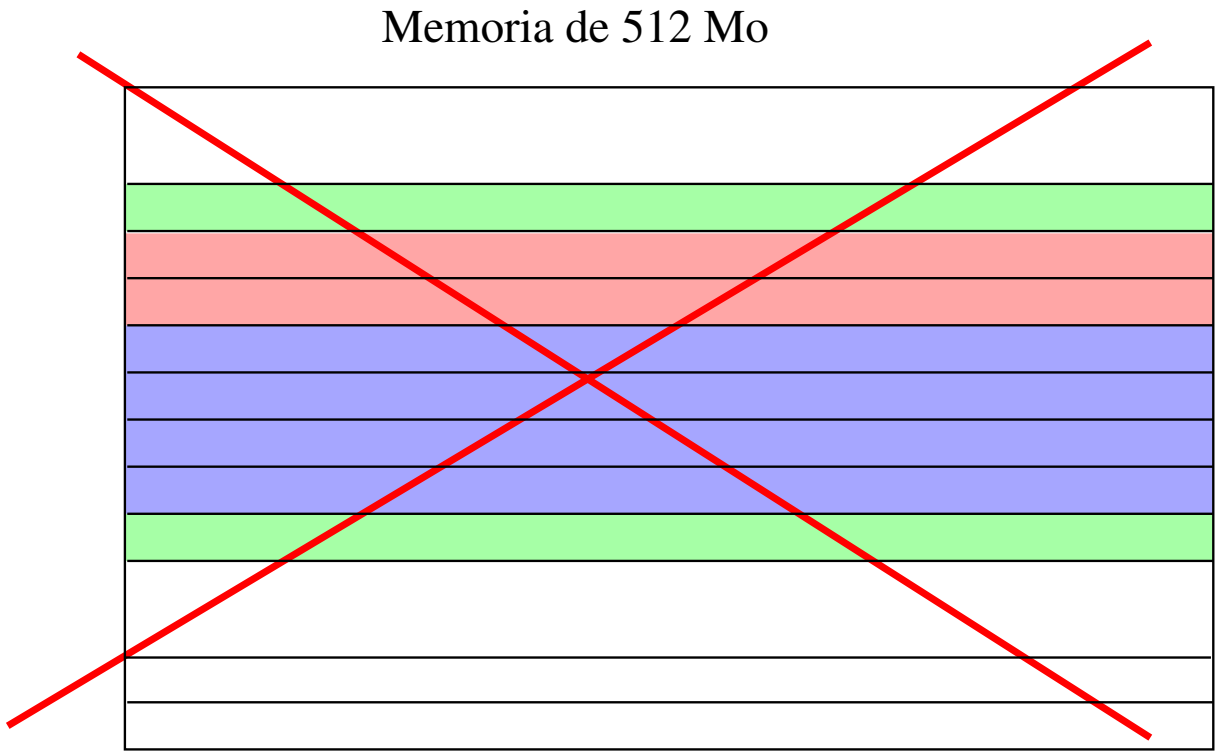
```
struct toto {
char c;
int i;
short s;
char o;
};
```



Tamaño de la estructura ?

Alineamiento de datos

```
struct toto {
char c;
int i;
short s;
char o;
};
```



No es así después de la compilación !

Alineamiento de datos

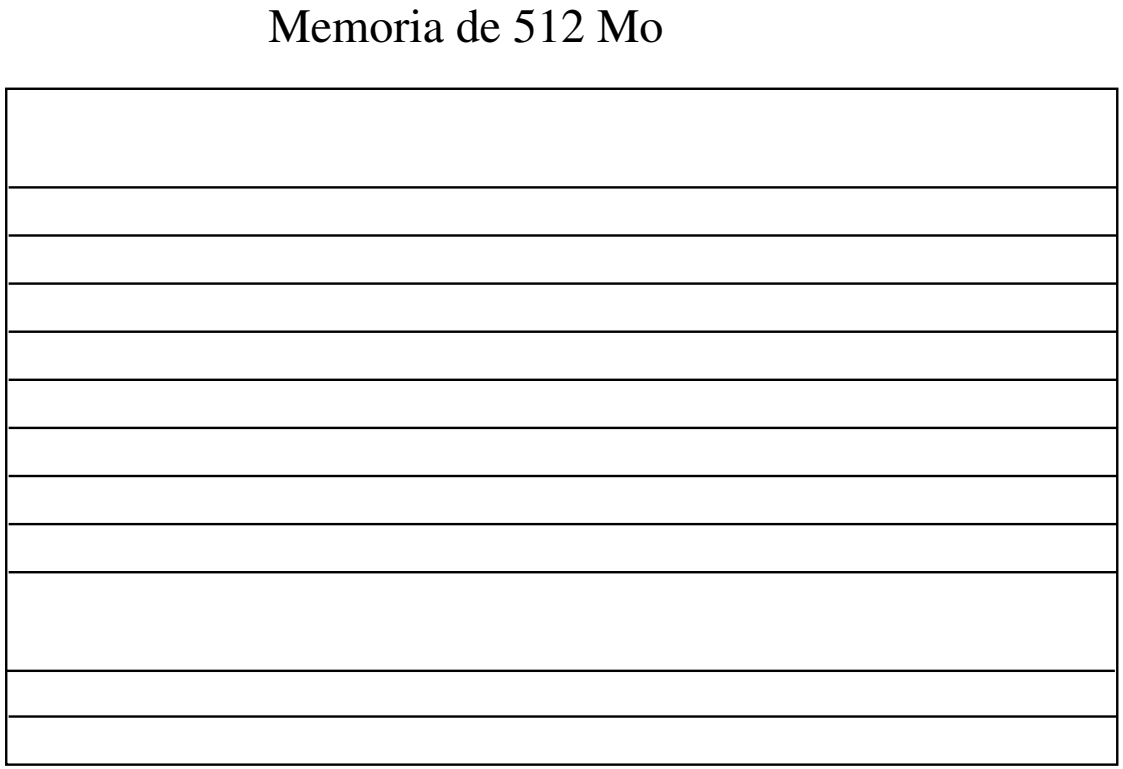
```
struct toto {
char c;
int i;
short s;
char o;
};
```

El compilador añade espacios de memoria no usados (pad). ¿Por qué ?

Alineamiento de datos

Como nosotros vemos
la memoria :

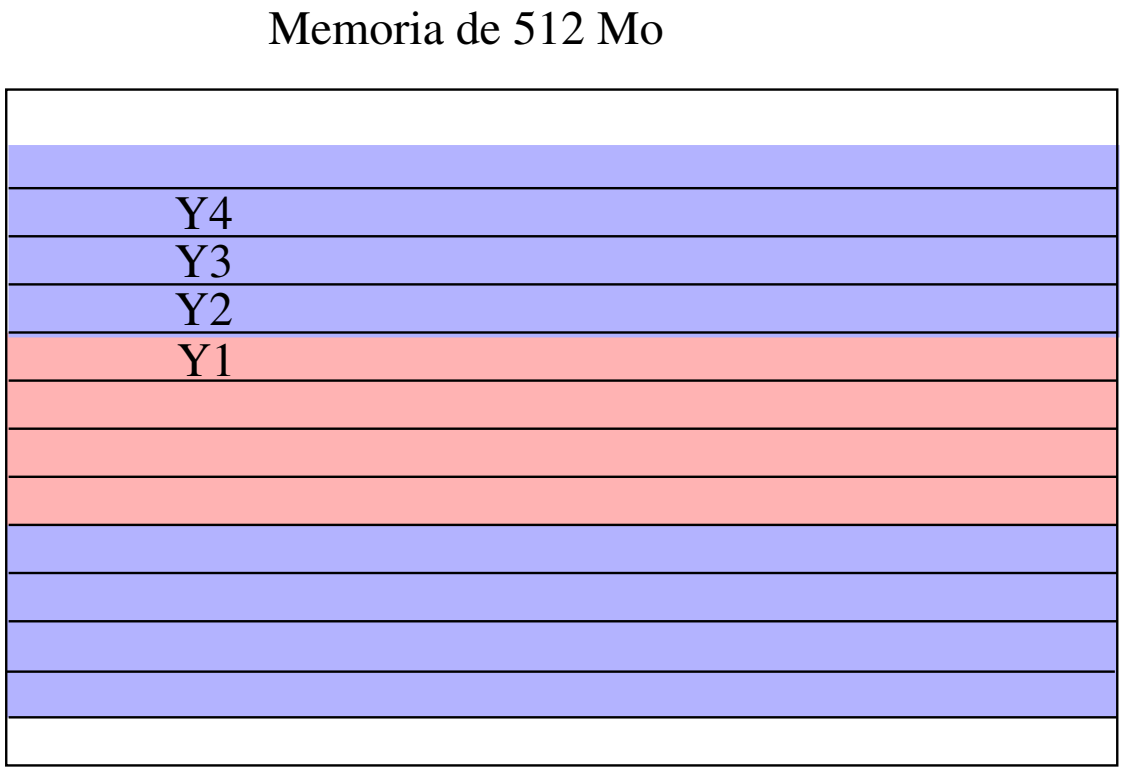
1 byte



Pensamos a una serie de bytes y ya.

Alineamiento de datos

Como la computadora ve la memoria :



GRANULARIDAD
(depende de la maquina)

La máquina accede a la memoria a través de un bus de un tamaño generalmente múltiplo de un byte. Puede ser 2 (procesadores 68k), 4 (maquinas 32 bits), 8 o 16.

Alineamiento de datos

- Unas máquinas aun negaban tratar datos no alineados (¡faltan los circuitos para hacerlo !)
- En todos casos, se desperdicia capacidades de procesamiento.
- Lo de las estructuras también vale al nivel de las variables locales.
- Dos niveles : intra-estructura y inter-estructura.

Mal alineamiento de datos : riesgos

Dependiendo de la arquitectura, del procesador, del OS,

- El programa es mas lento.
- La aplicación puede fallar.
- El OS puede fallar.
- La aplicación puede dar resultados erróneos.

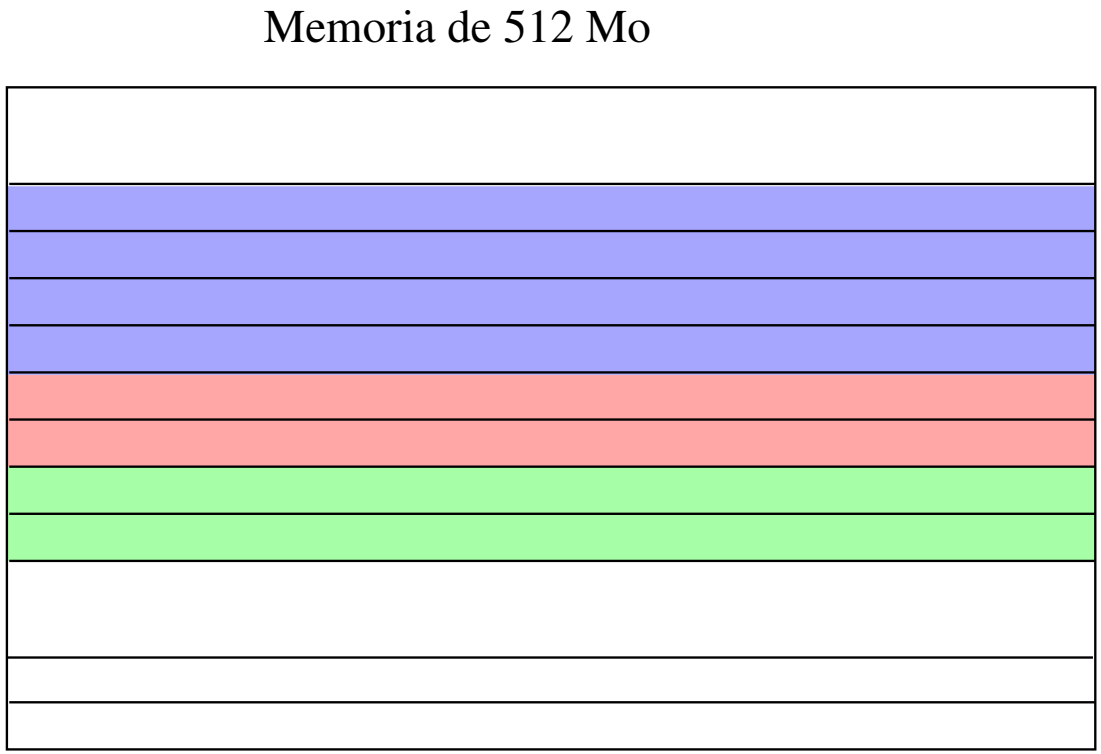
Alineamiento de datos

```
struct toto {
char c;
int i;
short s;
char o;
};
```

El compilador alinea los datos por sí mismo para facilitar la ejecución.

Alineamiento de datos

```
struct toto {
char c;
char o;
short s;
int i;
};
```



Pensar bien la organización de las estructuras ayuda.

Alineamiento de datos : reglas de padding

Sobre sistemas 32 bits Linux

- Un octeto único es alineado en cualquier dirección.
- Un short es alineado sobre direcciones pares.
- Un long es alineado sobre direcciones múltiplos de 4.
- Estructuras entre 1 y 4 octetos $\longrightarrow$ 4 octetos.
- Estructuras de mas de 4 octetos $\longrightarrow$ múltiplo de 4 octetos.

Pero hay reglas diferentes...En particular para el uso de los SIMDs (optimización) hay que alinear los datos sobre 16 bytes.

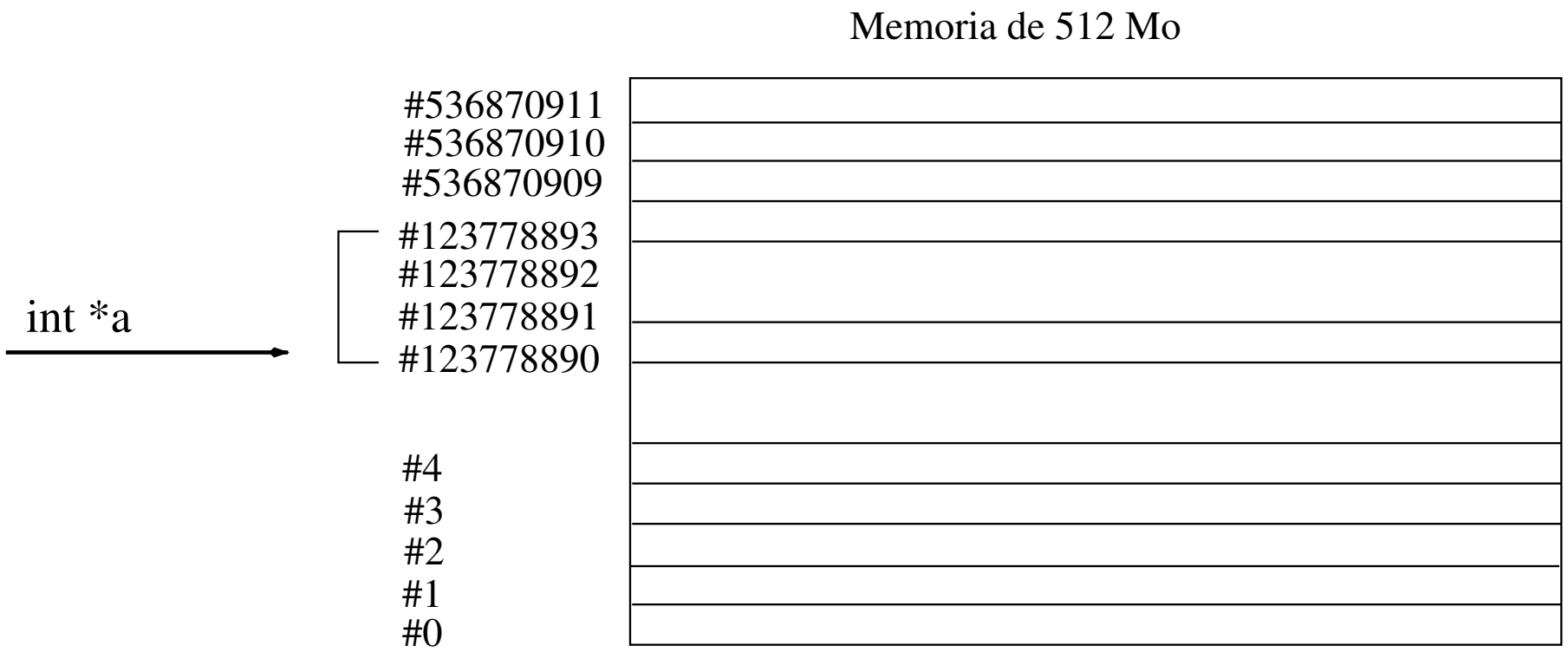
Outline

Definición simple

Un apuntador es un tipo que refiere a un dato en la memoria; incluye dos cosas :

1. La dirección en la memoria en que esta el dato considerado.
2. La manera de leer los datos a partir de este lugar de la memoria, i.e. que tipo de datos son.

Definición simple



- 123778890 es una dirección, sería la misma para un char *.
- Manipular el apuntador por operadores aritméticos se hará por paquetes de int (4 octetos).

Acceso al valor

Se hace por el operador unario de indirección * :

```
int a = 40;
int *a_ptr = &a;
*a_ptr = a*a;
```

Este operador tiene un nivel de prioridad 2 según la clasificación de la previa clase:

```
int a = 10;
int *b = &a;
int c = a**b**b;
printf("c = %d \n",c);
```

Acceso al valor

En caso de estructuras, existe un operador sinónimo de la combinación de $*$ y de . :

```
persona nery;
persona *neryPtr = &nery;
neryPtr->edad = 25;
(*neryPtr).edad = 25;
```

Las dos ultimas lineas son equivalentes.

Aritmética sobre apuntadores

Un apuntador es nada mas que un entero; entonces, se puede aplicar unos operadores aritméticos, los que sí tienen sentido :

- Adición de un entero; el resultado es un apuntador de mismo tipo, en un lugar trasladado de la memoria;
- Substracción de un entero; el resultado es un apuntador de mismo tipo, en un lugar trasladado de la memoria;
- Substracción entre dos apuntadores; regresa un entero (espacio memoria entre los dos apuntadores, en unidades correspondiendo al tipo del apuntador).

También se puede usar operadores de comparación, con la condición que se compare apuntadores de mismo tipo.

Aritmética sobre apuntadores

```
int i = 15;
int *p1, *p2;
p1 = &i;
p2 = p1 + 2;
printf("p1 = %ld \t p2 = %ld \n", p1, p2);
```

Si $p1 = 12358952$, $p2 =$?

Aritmética sobre apuntadores

```
double i = 15.0;
double *p1, *p2;
p1 = &i;
p2 = p1 + 2;
printf("p1 = %ld \t p2 = %ld \n",p1,p2);
```

Si $p1 = 12358952$, $p2 =$?

Aritmética sobre apuntadores

La utilización mas frecuente es para recorrer los datos de un arreglo,

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
int main()
{
   int *p;
   for (p = &tab[0]; p <= &tab[N-1]; p++)
      printf(" %d \n",*p);
   for (p = &tab[N-1]; p >= &tab[0]; p--)
      printf(" %d \n",*p);
   return 0;
}
```

Apuntadores : el operador []

El operador [] es equivalente a una combinación entre el operador aritmético '+' y el operador de indirección :

```
int tab[10];
int *p=&tab[0];
p[2]   = 5;
*(p+2) = 5;
```

Las dos últimas lineas son equivalentes.

Apuntadores y funciones

Para estructuras grandes, puede ser muy costoso pasar variables de este tipo a funciones, como parámetros (hay copias. . .), especialmente si usamos esta función muy seguido : en cambio, pasarle la dirección de esta estructura y su tipo (i.e. un apuntador) es suficiente.

Apuntadores y funciones

Comparar :

```
typedef struct dataStruct {
        double data[100];
        int width;
        int height;
} data;

void func1(data mydata) {
int w = mydata.width;
...
};
void func2(data *mydata) {
int w = mydata->width;
...
};
```

Apuntadores y funciones

Otro uso muy frecuente de los apuntadores es regresar el resultado de una función por el medio de un apuntador :

```
void esJoven(persona *cons,int respuesta) {
  if (cons->edad<30)
      respuesta = 1;
  else
      respuesta = 0;
};
int main() {
      persona nery; nery.edad = 25;
      int joven = 0;
      esJoven(&nery,joven);
      if (joven) printf("nery es un joven");
      else printf("nery ya es viejito");
      return 0;
}
```

Apuntadores y funciones

El programa no tiene el comportamiento que uno quería, porque por construcción, una función tiene un espacio de memoria particular a ella : copia los valores de sus parámetros en otras variables del mismo tipo. La solución es ¡pasarle un apuntador !

Apuntadores dobles

Un apuntador tiene un tipo dado, es un valor guardable en memoria, entonces, como para los otros tipos, podemos definir un apuntador hacia un apuntador :

```
int a = 1;
int *aptr = &a;
int **aptrptr = &aptr;

printf("La dirección de a es %ld y su valor es %d \n"
        *aptrptr,**aptrptr);
```

Se podría leer como $(int*)*$ para entenderlo mejor.

Apuntadores múltiples

De la misma manera se puede definir apuntadores múltiples : el apuntador de grado n es un apuntador, simple, que apunta hacia apuntadores (direcciones) de grado $n-1$.

Outline

Arreglos dinámicos

A veces no sabemos de antemano cual será el tamaño del arreglo que necesitamos para guardar datos. Dos opciones :

1. Usar un limite máximo y arreglo estático. El programa manejará todo.
2. Usar alocación dinámica : usando la función *malloc*, pedimos por una cantidad de memoria que esta reservada en la pila ; nos incumbe liberarla.

Arreglos estáticos : limites

Imagina un ejemplo en que queremos aplicar una operación elemento por elemento a un arreglo y guardar el resultado en otro arreglo :

```
#define N 100;
int *applyFunction(int *arregloIn) {
        int arregloOut[N];
        int i;
        for (i=0;i<N;i++)
            arregloOut[i] = ...;
        return &arregloOut[0];
}
int main() {
        int arreglo[N];
        int *result = applyFunction(arreglo);
        ...
}
```

Arreglos dinámicos

El arreglo dinámico sobrevive afuera de una función, no es el caso del arreglo estático (excepto si esta declarado con *static*). La función *malloc* toma como parámetro el numero de bytes requerido :

```
int *arregloDinamico =
        (int *)malloc(tamanoArreglo*sizeof(int));
```

No inicialización de los valores es hecha !

Arreglos dinámicos

Alternativamente, se puede usar la función calloc, que toma como parametro el numero de objetos de un tipo dado y el tamaño del tipo.

```
int *arregloDinamico =
        (int *)calloc(tamanoArreglo, sizeof(int));
```

calloc hace la inicialización de los arreglos a 0 !

Arreglos dinámicos

Un peligro : estamos pidiendo al sistema que nos reserve memoria; puede ser que no haya el espacio necesario : es importante verificar qué regresa *malloc* o *calloc*. En caso de que la alocación no funcioné, esas funciones regresan NULL :

```
int *arregloDinamico =
        (int *)calloc(tamanoArreglo, sizeof(int));
if (arregloDinamico == NULL)
        ...
```

Arreglos dinámicos

Finalmente, el espacio alocado se libera con la función free :

```
int *arregloDinamico =
        (int *)calloc(tamanoArreglo, sizeof(int));
....
free(arregloDinamico);
```

P.D. : la única referencia al espacio alocado es el apuntador regresado por el *malloc* o el *calloc*, es importante no perderlo para usar el *free*.

Arreglos dinámicos multiples

```
int **arregloDinamicoDoble =
        (int **)calloc(tamanoArregloX, sizeof(int *));
if (arregloDinamicoDoble==NULL)
        ...
for (i=0;i< tamanoArregloX; i++) {
        arregloDinamicoDoble[i] =
        (int *)calloc(tamanoArregloY, sizeof(int));
        if (arregloDinamicoDoble[i]==NULL)
        ...
}
....
for (i=0;i< tamanoArregloX; i++)
          free(arregloDinamicoDoble[i]);
free(arregloDinamicoDoble);
```