Programación en C

Manipulación de memoria

Manipulación de memoria

Todas las operaciones que realiza el CPU tienen como objetivo ocasionar algún cambio en memoria. Necesitamos memoria para leer operandos y para guardar resultados.

A diferencia de otros lenguajes de más alto nivel, en C podemos interactuar directamente con la memoria. Esto se logra mediante el uso de **pointers**.

Los pointers nos dan la capacidad de saber dónde se encuentra un objeto en memoria, así también como leer o escribir cualquier posición en memoria que deseemos.

Pointers

C99: “Objeto cuyo valor brinda una referencia a una entidad del tipo referenciado”.

```
char* a; // pointer to char
```

El estándar no pone ningún requerimiento sobre cómo se implementa.

En la mayoría de los casos, un pointer es una dirección de memoria.

Por convención:

```
char* a; -> char *a;
```

C99 6.2.5.20

Pointers

RAM significa “memoria de acceso aleatorio”. Es decir, podemos acceder a cualquier valor en ella en la misma cantidad de tiempo, sin importar dónde esté (O(1)).

```
char *a = 1; // a points to 01101000
```

Cada dirección de memoria guarda 1 byte (8 bits).

Para expresarlas de una manera más compacta, las direcciones se escriben en **hexadecimal**.

0, 1, 2, …, 9, a, b, c, d, e, f

Memoria

1	01101000
2	01101001
3	00100001
4	00000000

Ex: 1.1, 1.2, 1.3

Null pointer

El null pointer es simplemente un pointer que garantiza:

- Ser distinto a cualquier pointer a un objeto
  - Pedir la referencia (&) de un objeto **nunca** puede dar un null pointer.
- Ser igual a cualquier otro null pointer
  - No importa el tipo que el pointer referencie (char, int, long, etc).

Desreferenciar un null pointer es undefined behavior. En la mayoría de los casos podemos esperar un error, pero no es garantizado.

Ex: 1.4 | C99 6.3.2.3.3/4, 6.5.3.4

Pass-by-value

En C, todas las funciones reciben parámetros como valor (pass-by-value).

Pass-by-value: la función recibe una copia del valor de los parámetros.

- Una función no puede cambiar el valor de sus parámetros fuera de su contexto.
  - No side-effects.

Ex: 2.1

Pass-by-reference

La contracara de pass-by-value es pass-by-reference.

Pass-by-reference: la función recibe una referencia al objeto usado como parámetro.

Podemos efectivamente hacer que C use pass-by-reference pasando un pointer como parámetro en vez del valor.

C sigue usando pass-by-value, pero el valor que se copia es el **pointer**.

Parámetros de salida

C no tiene excepciones, necesitamos una manera de manejar casos de error.

Opción 1: retornar un valor fuera del rango de valores válidos y tomarlo como indicación de error (ej: indexOf == -1).

Opción 2: retornar un struct que incluya el resultado y un código de error.

```
struct res { int val; int error; };
```

Opción 3: retornar el valor del error y recibir un pointer a dónde guardar el resultado como parámetro (pass-by-reference).

La 3era es la más usada.

Ex: 3.1, 3.2, 3.3

Pass-by-value: copias

En los casos en que no requerimos múltiples outputs de nuestra función es totalmente lógico retornar el resultado normalmente. No hay necesidad de modificar un valor externo mediante pointers.

Sin embargo:

Pass-by-value: la función recibe una **copia** del valor de los parámetros.

- El objeto entero tiene que ser copiado
  - Más memoria
  - Más tiempo

¿Cuánto estamos copiando?

`sizeof`: operador que devuelve la cantidad de bytes que ocupa un objeto. Retorna un **`size_t`** que es un número entero positivo.

```
size_t char_size = sizeof(char);
```

El resultado de **`sizeof`** depende de la arquitectura para la cual compilemos el programa.

De la misma manera, el valor máximo que puede tomar un **`size_t`** también depende de la arquitectura.

Ex: 4.1, 4.2, 4.3, 4.4

Structs

Si hablamos de tipos primitivos (int, long, double, etc), pasar un pointer como argumento generalmente termina copiando más

Ejemplo: 4 bytes por int vs 8 bytes por pointer.

Sin embargo, un struct puede ser arbitrariamente grande y el costo de copiarlo cada vez que llamamos a una función puede ser considerable.

Salvo casos especiales es más eficiente pasar un pointer al struct.

Además, pasar un pointer es necesario si queremos modificar sus elementos.

Ex: 5.1, 5.2

Java: pass-by-value o pass-by-reference?

Cuando pasamos un tipo primitivo a una función de Java no podemos modificar su valor original. Esto nos dice que Java usa pass-by-value.

Pero si pasamos un objeto si podemos modificarlo. ¿Puede ser que Java use pass-by-reference o pass-by-value dependiendo el tipo del parámetro?

No, Java siempre usa pass-by-value.

La diferencia es que Java siempre usa referencias (pointers) a objetos. La desreferenciacion del pointer es implícita.

Nunca trabajamos con los “verdaderos objetos” en Java

Ex: 5.3

Arrays

```
char word[4] = {‘d’, ‘u’, ‘c’, ‘k’};
```

Como en Java, podemos leer o escribir en la posición que queramos del array.

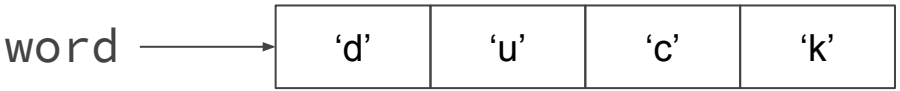
```
char d = word[0]; // d

char c = word[2]; // c

word[3] = ‘t’; // word is now ‘duct’
```

Ex: 6.1

Arrays

C99: “Set de objetos del **mismo tipo** alocados **contiguamente** en memoria”.

El tipo “array” no existe per se. En el ejemplo la variable `word` es en realidad de tipo **`char*`**. Es un pointer al primer **`char`** en el array.



El próximo **`char`** se encuentra inmediatamente después en memoria.

Arrays

`word` es un pointer, así que también lo podemos desreferenciar.

```
char d = *word; // d
```

Desreferenciar un pointer es lo mismo que agarrar el primer índice de un array. ¿Cómo funcionan los otros índices?

Ex: 6.3

Pointer arithmetic

```
??? *var; // pointer to something
```

```
var[n] == *(var + n);
```

`var[n]` no es más que decir “quiero el valor en memoria con un offset de `n` desde el pointer `var`”.

Que significa “offset de `n`” depende del tipo del pointer. Más específicamente, depende de su tamaño (**`sizeof`**) .

Ejemplo: un offset de 1 en **`char`** es 1 byte. En un **`int`** 4 bytes.

Ex: 6.4

Pointer arithmetic

```
??? *var; // pointer to something

??? *next = var + 1;

??? *up_5 = var + 5;

??? *prev = var - 1;

??? *down_5 = var - 5;

var++; var--; // also ++var, --var

ptrdiff_t offset_dif = var - other_ptr;
```

Ex: 6.5, 6.6, 6.7

What even is IndexOutOfBoundsException?

Si una **variable** “array” no es más que un pointer, no tenemos manera de chequear cuando estamos trabajando fuera de los límites del array.

Nada impide pedir el elemento 200 de un array de tamaño 5.

Nada impide indexar cualquier pointer al azar, esté conectado a un array o no.

Nada impide indexar un array con un número negativo.

Todos estos (y muchos más) son **undefined behavior**.

C99 6.5.2.1.2, 6.5.6.8

Arrays

Al trabajar con arrays siempre necesitamos saber su tamaño de alguna manera. Esto puede ser:

- Una variable con el número de elementos.
- Indicando el final mediante un elemento especial. Por ejemplo NULL.

Ex: 6.8, 6.9

Arrays != pointers

Aunque la **variable** que declaramos al armar un array funciona para los propósitos del resto del código como un pointer, **no son lo mismo**.

Declarar un array implica que obtenemos una alocación de memoria y un pointer al inicio de esa alocación.

```
char word[10];
```

word →

```
char *word;
```

word →

Ex: 6.10

Strings

En C un String no es más que un array de **char** con el final marcado con un ‘\0’.

El ‘\0’ es un char especial llamado **null terminator**. Corresponde al 0 en ASCII.

```
char str[5] = {‘d’, ‘u’, ‘c’, ‘k’, ‘\0’};

char str[] = “duck”; // still 5 chars

char *str = “duck”; // equivalent (almost)
```

Strings

La librería estándar brinda funciones para trabajar con strings en el header **`string.h`**. (Google / consola: man string)

Todas las funciones asumen un string bien construido (con null terminator).

Los argumentos de las funciones solo pueden especificar un **`char*`**.

¿Qué pasa si le damos un array de **`char`** que no tenga null terminator?

Undefined behavior! Los loops de la librería van a continuar trabajando hasta encontrar algo equivalente a `'\0'` (un byte de ceros).

Ex: 7.2, 7.3

Strings

¿Cómo conseguiría un string sin null terminator? Muy fácil.

Los arrays son de tamaño fijo, pero podemos recibir un string de cualquier longitud:

- Input del usuario
- Leer un archivo
- Variabilidad de un algoritmo
- etc.

Esto sumado a errores de programación = undefined behavior.

Ex: 7.4, 7.5, 7.6, 7.7

Structs

Arrays según C99: “Set de objetos del **mismo tipo** alocados **contiguamente** en memoria”.

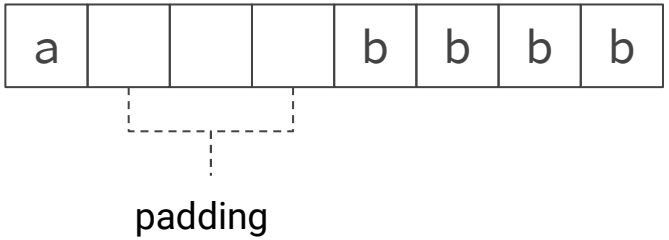
Structs según C99: “Set de objetos, posiblemente de **distintos tipos**, alocados **secuencialmente** en memoria”.

Los structs son capaces de contener distintos tipos de datos juntos, pero limitaciones del hardware obligan a introducir **padding** entre elementos en ciertos casos.

C99 6.2.5.20

Structs

```
struct data { char a; int b; };
```



Generalmente, un tipo primitivo de tamaño **n** solo puede vivir en direcciones de memoria **múltiplo de n**.

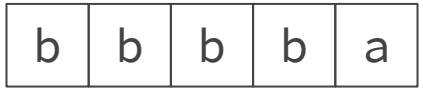
Si esto no pasa: undefined behavior.

Ex: 8.1

Structs

Podemos reordenar los miembros del struct, de manera que se encuentren alineados.

```
struct data { int b; char a; };
```



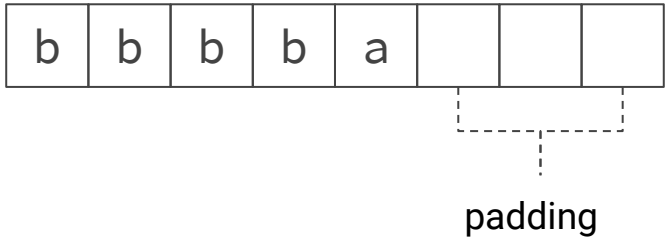
Esto es una manera fácil de ahorrarse 3 bytes. Si tenemos muchas instancias del `struct data` la diferencia puede ser considerable.

Ex: 8.2

Structs

En realidad:

```
struct data { int b; char a; };
```



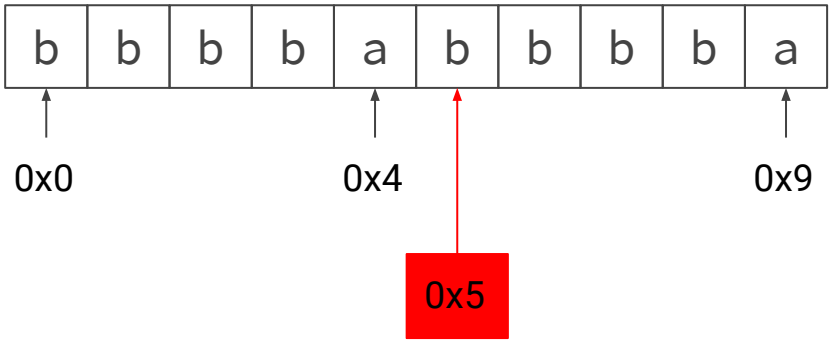
En este caso en particular siempre vamos a tener padding, no importa el orden de los elementos

Ex: 8.3

Structs

Esto se debe a lo que pasaría en un array:

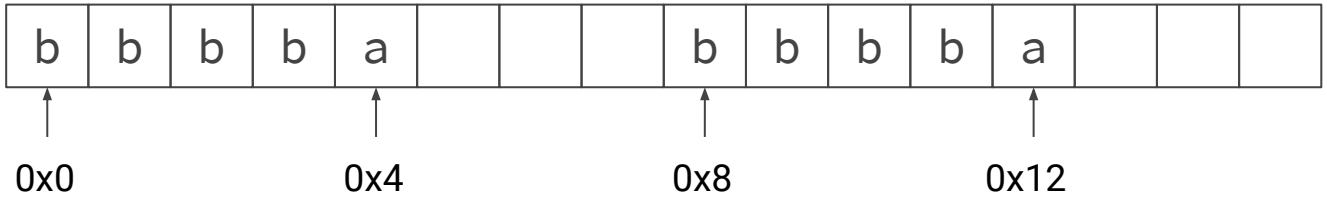
```
struct data { int b; char a; };

struct data array[2];
```



Structs

Con padding todos los elementos están alineados:

```
struct data { int b; char a; };
struct data array[2];
```



Por supuesto, acceder a un valor del padding es undefined behavior.

Ex: 8.4

“Polimorfismo” con structs

Como la estructura en memoria de un struct depende sólo del tipo y orden de sus miembros, podemos tener situaciones como ésta:

```
struct base_data {
    char a;
    int b;
};
```

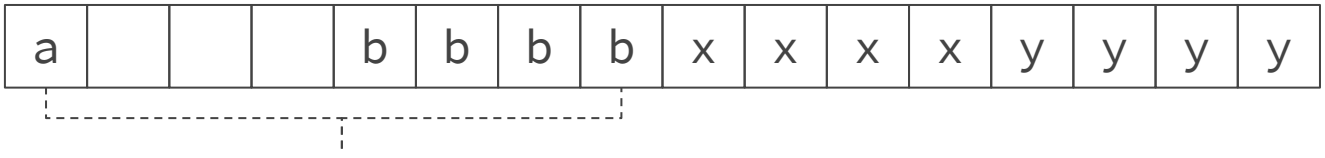
```
struct extra_data {
    char a;
    int b;
    int x;
    int y;
};
```

“Polimorfismo” con structs

En memoria vamos a tener:

struct base_data

struct extra_data



“Polimorfismo” con structs

Como la **sección inicial** de los structs es la misma, todo lo que utilice un `struct base_data*` puede tranquilamente recibir un `struct extra_data*`.

La clave es que deben ser **pointers**, ya que si utilizaramos el verdadero valor del struct un `base_data` ocupa menos espacio que un `extra_data` y surgen problemas al alocar memoria. Undefined behavior!

Otro punto importante es que solo importa el **tipo** de los miembros de los structs y su **orden**. El nombre de los miembros del struct no afecta nada.

Si el orden o el tipo es distinto => undefined behavior.

Ex: 8.5, 8.6, 8.7

Unions

C99: “Un set superpuesto de miembros, opcionalmente con nombres y distintos tipos”

```
union a_or_b { char a; int b; };
```

`sizeof` de una union = max **`sizeof`** de sus miembros.

Alineación de una union = max alineación de sus miembros.

Unions

Efectivamente, el tipo de la union depende del último valor que se le asignó.

Leer un miembro que no fue el último escrito = undefined behavior.

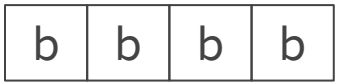
```
union a_or_b x;
```

```
x.a = 'm';
```

union a_or_b

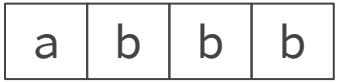

```
x.b = 123456;
```

union a_or_b




```
x.a = 'w';
```

union a_or_b



(probablemente)

Ex: 9.2

“Polimorfismo” con structs y unions

Si tenemos una union con structs que tienen la misma **sección inicial**, el standard garantiza que, si la union actualmente contiene cualquiera de esos structs asignados, es válido acceder a cualquier valor dentro de la **sección inicial**.

```
union d {
  s base_data a;
  s extra_data b;
};
```

```
s base_data {
  char a;
  int b;
};
```

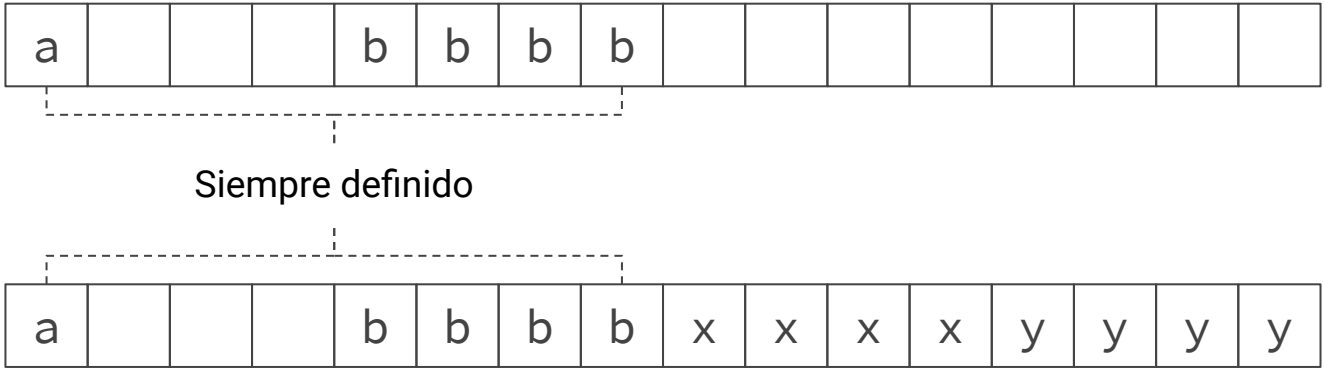
```
s extra_data {
  char a;
  int b;
  int x;
  int y;
};
```

s == struct (para el slide, no en C)

C99 6.5.2.3.5

“Polimorfismo” con structs y unions

```
union d { s base_data a; s extra_data b; };
```



Ventaja: no es necesario usar pointers.

Desventaja: mayor uso de memoria.

Ex: 9.3