



Oradores

Matías Dumrauf
Facundo Viale

- Presentación/Quiénes somos?
- Objetivos de las Talks
- Web: <http://sites.google.com/site/utnfrbactalks/>
- Group: <http://groups.google.com.ar/group/c-talks>
- Repositorio SVN: <http://code.google.com/p/c-talks>

¿Por qué C ?

- Mediano-Bajo Nivel y Bajo consumo de recursos
 - Mayor aprovechamiento del Hardware y la Arquitectura.
 - Mayor interacción con el OS.
- Muy usado en Sistemas donde se requiere:
 - Alta disponibilidad de los recursos.
 - Funcionamiento en tiempo real.
 - Monitoreo de Hardware.
 - Alta Performance.
- Actualmente muy utilizado en proyectos Open Source que involucran desarrollo de:
 - Kernel & Drivers.
 - Maquinas Virtuales.
 - Gestor de ventanas, Escritorios, etc ...
 - Versionadores de Código.
 - Librerías Gráficas.
 - Aplicaciones multimedia: Codecs, Reproductores, Editores, etc ...
- Puede implementarse “cualquier cosa”, pero no es conveniente para “cualquier cosa”.

Talk I

Basis & Best Practices

Temario

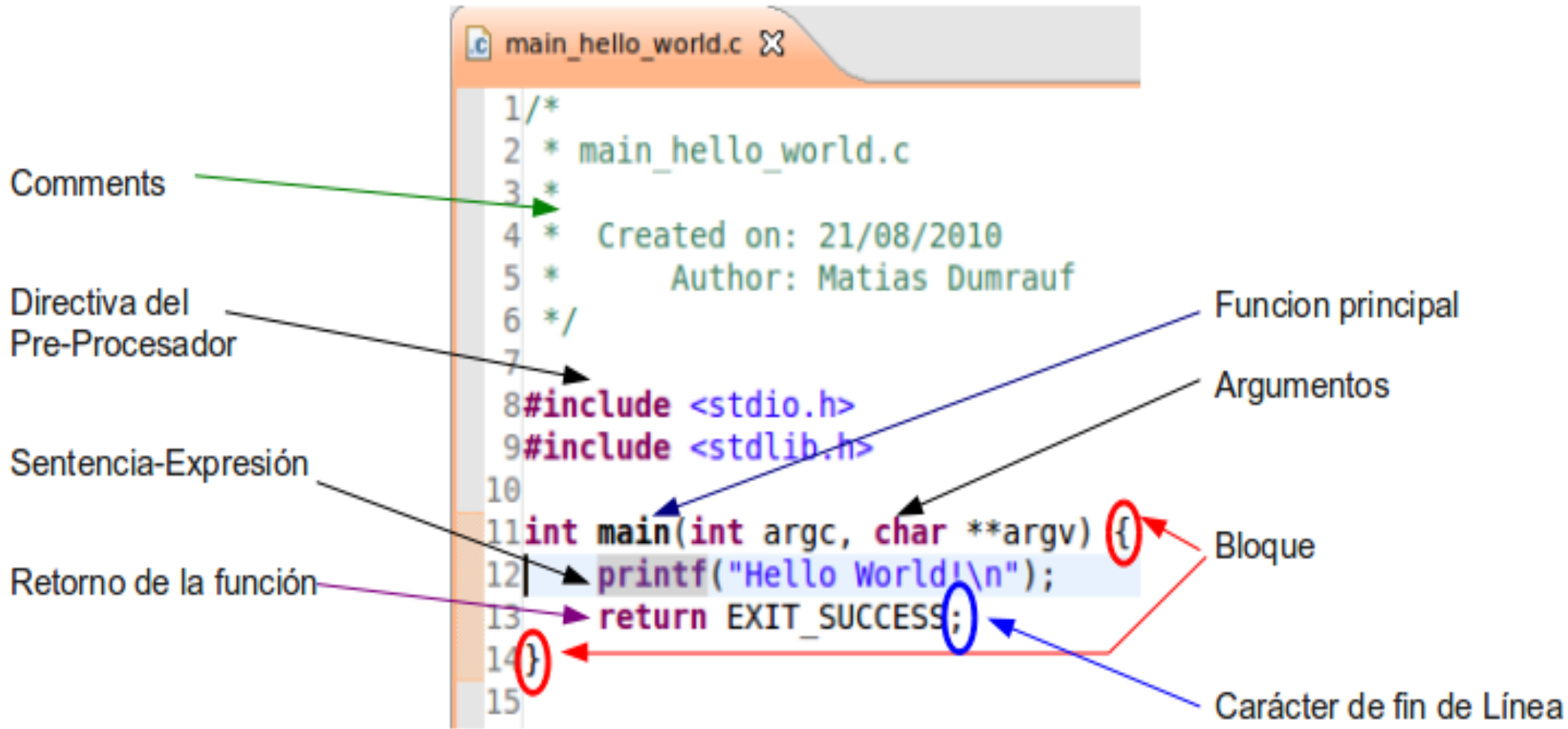
- ANSI C for Dummies
- TAD
- Best Programming Practices
- Casos de aplicación complejos

ANSI C for Dummies

Mi primer programa en ANSI C

```
.c main_hello_world.c X
1/*
2 * main_hello_world.c
3 *
4 * Created on: 21/08/2010
5 * Author: Matias Dumrauf
6 */
7
8#include <stdio.h>
9#include <stdlib.h>
10
11int main(int argc, char **argv) {
12    printf("Hello World!\n");
13    return EXIT_SUCCESS;
14}
15
```

Mi primer programa en ANSI C



Mi primer programa en ANSI C

- Todo programa tiene un main
- El main es una función (la "función principal")
- Recibe argumentos (argc, argv, para los programas comando) y retorna un valor (resultado de la ejecución)
- La lógica de un programa está dentro del main

Tipos Primitivos de Datos

En arquitecturas de 32 bits:

		Tipo	Tamaño	Rango
Enteros {		char	1 byte (8 bits)	(-2^7 ; $2^7 - 1$)
		int	4 bytes (32 bits)	(-2^{31} ; $2^{31} - 1$)
Reales {		float	4 bytes (precisión simple)	$1.18e-38 \leq X \leq 3.40e38$
		double	8 bytes (precisión doble)	$2.23e-308 \leq X \leq 1.79e308$

Tipos de Datos

- Un Tipo de Dato declara el tipo, el tamaño en memoria y el rango, de una variable.
- C es un lenguaje medianamente tipado.
- Se puede acotar/extender el **rango** de un tipo de dato entero usando Calificadores de Tipo:

De Longitud:

short

y **long** (ej. **short int**, **long int**)

De Signo: **signed** y **unsigned**

-

short

y **long** pueden omitir el **int** y utilizarse como Tipos de Datos.

Ej:

short

a; es lo mismo que **short int** a;

Siempre: **short** < **int** < **long**

En arquitecturas de 32 bits (algunos tipos):

Tipo	Tamaño	Rango	Uso
singed char	1 byte (8 bits)	(-2^7 ; $2^7 - 1$)	Numeros pequeños, caracteres ASCII
unsigned char	1 byte (8 bits)	(0 ; 255)	Numeros pequeños, ASCII Extendida
short int ó short	2 bytes (16 bits)	(-2^{15} ; $2^{15} - 1$)	Indices y numeros pequeños
int	4 bytes (32 bits)	(-2^{31} ; $2^{31} - 1$)	Indices, numeros pequeños
long int ó long	4 bytes (32 bits)	(-2^{31} ; $2^{31} - 1$)	Numeros grandes
unsigned long	4 bytes (32 bits)	(0 ; 2^{32})	Distancias astronómicas
float	4 bytes (precisión simple)	$1.18e-38 \leq X \leq 3.40e38$	Precisión científica (7-dígitos)
double	8 bytes (precisión doble)	$2.23e-308 \leq X \leq 1.79e308$	Precisión científica (15-dígitos)
long double	8 bytes (precisión doble extendida)	$3.37e-4932 \leq X \leq 1.18e4932$	Precisión científica (18-dígitos)

- El Tipo Entero básico es **int**.

- **char** también es un entero! Como ocupa 1 B, el rango es menor. Por lo tanto:

char < short < int < long

- Tipos

const: para declarar variables constantes.

Ej: **const float** pi = 3,14159;

Permitido:

printf("El valor de Pi es: %.5f\n", pi);

No Permitido:

pi = 3,1416;

- Definición de variables al inicio de un bloque, de la forma:

<tipo_dato> <nombre_variable> ;

Ej:

int a;

- Operadores básicos:

Asignación:

a = 4;

a += 2;

a -= 2;

Igualdad:

```
while( a == 1 ){  
    puts("siempre");  
    puts("la tenes adentro");  
}
```

```
if( a != b )  
    printf("No sos igual");
```

Lógicos:

```
if( (a == 1) && (b != 3) )  
    puts("Se dio la condicion AND");
```

```
if( (a == 1) || (b != 3) )  
    puts("Se dio la condicion OR");
```

```
int a = 0;
```

```
if( !a ) puts("Negamos a");
```

Matemáticos:

+ y - : suma y resta

*** y /** : multiplicación y división

% : modulo o resto

> y < : mayor y menor

- Valores de Verdad:

- 1: Verdadero
- 0: Falso

- Todas las Expresiones en C tienen un Valor de Verdad, por más que no sean booleanas. Por defecto, el Valor de Verdad es **Verdadero**, a menos que el tipo de expresión indique lo contrario.

Un ejemplo de una expresión que no siempre resulta tener un valor de verdad 1 (verdadero) es la **Expresión Condicional**:

<exp_cond> ? <sentencia1> : <sentencia2> ;

Ej:

int valor = (**0 == 4**) ? 1 : 0; -----> se puede prescindir del uso de ()
por el orden de evaluación.

- Condicones/Selecciones:

El **if** siempre 'pregunta' por distinto de cero.

simple:

```
( <expresion> ){  
  <sentencial1>;  
  <sentencia2>;  
  .....  
  <sentenciaN>;  
}
```

if Anidados:

```
if( <expresion> ){  
  <sentencial1>;  
  .....  
  <sentenciaN>;  
}else if{  
  <sentencia1>;  
  .....  
  <sentenciaN>;  
}
```

if/else:

```
if( <expresion> ){  
  <sentencial1>;  
  .....  
  <sentenciaN>;  
}else{  
  <sentencia1>;  
  .....  
  <sentenciaN>;  
}
```

switch/case:

```
switch( <expresionEntera> ){  
  case <exp_const1> :  
    <sentencial1>;  
    .....  
    <sentenciaN>;  
    break;  
  
  case <exp_constN> :  
    <sentencial1>;  
    .....  
    <sentenciaN>;  
    break;  
  
  default:  
    <sentencial1>;  
    .....  
    <sentenciaN>;  
    break;  
}
```


- Iteraciones:

El **while** es un caso particular del **for**.

```
while( <expresion> ){  
    <sentencia1>;  
    <sentencia2>;  
    .....  
    <sentenciaN>;  
}  
  
do{  
    <sentencia1>;  
    <sentencia2>;  
    .....  
    <sentenciaN>;  
} while( <expresion> );
```

```
int i;  
  
for( i = 0; <expresion>; ++i ){  
    <sentencia1>;  
    <sentencia2>;  
    .....  
    <sentenciaN>;  
}
```

- Ejemplos de código.

Standard Library C89

Headers	Functions
<assert.h>	<i>assert()</i>
<ctype.h>	<i>int iscntrl(int); int isspace(int); int isdigit(int); int tolower(int); int toupper(int); etc ...</i>
<errno.h>	The <errno.h> header shall provide a declaration or definition for errno.
<float.h>	Contains macros that expand to various limits and parameters of the standard floating-point types.
<limits.h>	Includes definitions of the characteristics of common variable types.
<locale.h>	<i>struct lconv* localeconv(void); char* setlocale(int, const char*); etc ...</i>
<math.h>	<i>double asin(double); double ceil(double); double log(double); double tan(double); etc ...</i>
<setjmp.h>	<i>void longjmp(jmp_buf, int); void siglongjmp(sigjmp_buf, int);</i>
<signal.h>	<i>int raise(int sig); void* signal(int sig, void (*func)(int))</i>
<stdarg.h>	<i>void va_start(pvar); type va_arg(pvar, type); void va_end(pvar)</i>
<stddef.h>	Defines the macros NULL and offsetof as well as the types ptrdiff_t, wchar_t, and size_t.
<stdio.h>	<i>FILE *fopen(const char *restrict, const char *restrict); int fclose(FILE *); int printf(const char *restrict, ...); int putc(int, FILE *); int remove(const char *); int scanf(const char *restrict, ...); etc ...</i>
<stdlib.h>	<i>void exit(int); void free(void *); void *malloc(size_t); long random(void); int atoi(const char *); etc ...</i>
<string.h>	<i>void *memset(void *, int, size_t); int strcmp(const char *, const char *); char *strcpy(char *restrict, const char *restrict); char *strtok_r(char *restrict, const char *restrict, char **restrict); etc ...</i>
<time.h>	<i>clock_t clock(void); char *ctime(const time_t *);</i>

Manejo de memoria en C

C es un lenguaje de muy bajo nivel por lo que hay que considerar lo siguiente:

- Pensá en como funciona el CPU
- Pensá en como funciona la RAM

Esto quiere decir, no importa si es int, char, puntero, arrays, estructuras, etc ... a fin de cuenta todos son bytes.

Para el procesador el tipo de dato, define de a cuantos bytes tiene que manejarse para una variable.
(Rango de la variable)

Para el programador, el tipo de dato no solo nos define un rango y un tamaño sino una forma de representar al dato.

Ej:

int a = 100; DEC 100 = BIN 0110 0100
char b = 'A'; 'A' = DEC 65 = BIN 0100 0001

Tamaño de un **int** en memoria 4 bytes.
Tamaño de un **char** en memoria 1 byte.

Memoria para "char b"

0x0FFF		...
0x1000		0100 0001
0x1001		...
0x1002		...

Memoria para "int a"

0x1FFF		...
0x2000		0000 0000
0x2001		0000 0000
0x2002		0000 0000
0x2003		0110 0100
0x2004		...

Manejo de memoria en C

Asignación Estática:

Consiste en el proceso de asignar memoria en tiempo de compilación. Es decir, cuando compilar cuando leer el código tomas las variables globales que encuentra en este y les asigna un espacio fijo dentro del binario generado. Este tipo de asignación es parte de lo que se conoce como Memoria Estática.

```
int a; /* Declarada fuera del main o cualquier función */
```

Asignación Automática:

Consiste en el proceso de asignar memoria en tiempo de ejecución, correspondiente a las variables locales a un bloque. Esto es gestionado automáticamente, cuando se entra en un bloque las variables locales son colocadas en un stack y cuando se sale del bloque son des-asignadas automáticamente.

<pre><i>void foo(){</i> <i>int a;</i> <i>}</i></pre>	<p><u>NOTA:</u> También son variables automáticas los propios argumentos de la función:</p>	<pre><i>void foo(char *s, int b){</i> <i>}</i></pre>
--	--	---

Asignación Dinámica:

Consiste en el proceso de asignar memoria en tiempo de ejecución, con la diferencia de que tanto la asignación como la des-asignación se realiza de manera explícita por el programador.

```
int *a; /* "Declarada dentro o fuera del main o cualquier otra función" */
```

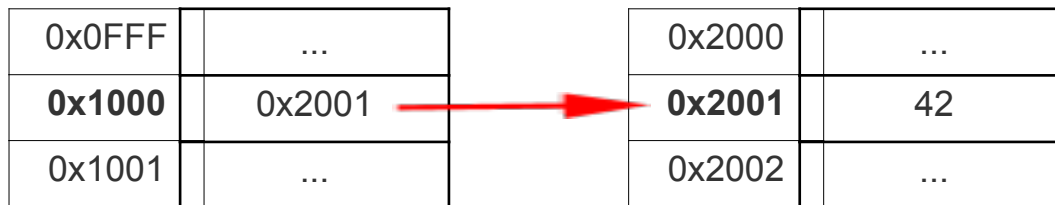
Manejo de memoria en C

Puntero:

Un puntero es una variable que referencia una región de memoria; en otras palabras es una variable cuyo valor es una dirección de memoria. Como los punteros son direcciones de memoria, su tamaño depende de la capacidad de direccionamiento del CPU que para arquitectura de 32 bits es justamente 32bits o 4 bytes o un unsigned int

En C cuando declaramos una variable como puntero colocamos el operado * delante de esta:

```
int *aux;
```



- La variable "aux" esta en la dirección 0x1000
- La variable "aux" contiene como valor la dirección 0x2001
- La variable "aux" apunta a la dirección 0x2001 la cual contiene el valor 42

C nos provee 2 operadores especiales para trabajar con punteros & y *, además de estos C nos permite operar con punteros como si se trataran de enteros (lo cual son) por lo que podemos sumar, restar, multiplicar y dividir.

- * - Nos devuelve el contenido de un puntero.
- & - No vuelve la dirección de memoria de una variable.

Para utilizarlos simplemente se coloca delante de la variable:

```
aux = Esto devuelve 0x2001
&aux = Esto devuelve 0x1000
*aux = Esto devuelve 42
```

Segmentation Fault: Es cuando se trata de acceder a una sección de memoria en la cual no tenemos permisos. Esto ocurre cuando un puntero apunta a una dirección fuera de rango o dentro de una rango el cual no es accesible.

```
int *aux = 0xFFFFFFFF;
*aux;
```

Manejo de memoria en C

Memoria Dinámica

- Es algo gestionado por el programador
- Su scope o alcance no esta limitado a un bloque en particular
- Nos permite mas versatilidad, ya que manejamos la memoria en tiempo de ejecución.

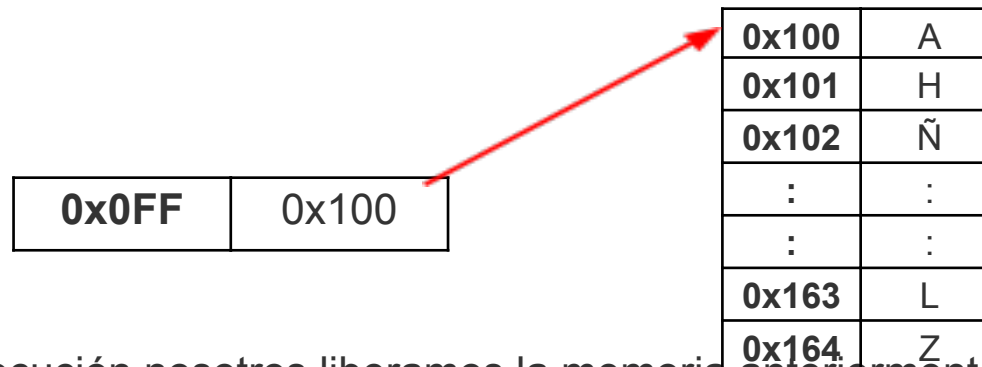
Pedir/Reservar Memoria: En tiempo de ejecución solicitamos, que se nos entregue una X cantidad de memoria

```
void *buff = malloc( 100 );
```

buff => 0x100

&buff => 0x0FF

*buff => 'A'



Liberar Memoria: En tiempo de ejecución nosotros liberamos la memoria anteriormente reservada. Esta queda sin uso y posteriormente se nos puede volver a entregar, completa o parcialmente, en caso de que solicitemos nuevamente mas memoria dinámica

```
free( buff );
```

Memory Leak: Es cuando un bloque de memoria reservada no es liberada. Esto es un error cometido por el programador al omitir la liberación de memoria, ya sea por olvido o por otro motivo. Este concepto comúnmente implica que se pierde la referencia a ese bloque de memoria reservado.

Arrays

Es un zona de almacenamiento contigua, que contiene elementos del mismo tipo.

```
char valore[10];  
int *valores[10];
```

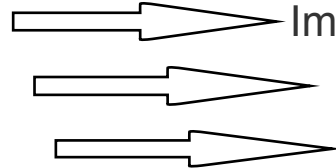
Inicialización:

```
char cadena[] = "Hola Mundo!";  
int secuencia[] = { 1, 2, 3, 4, 5, 6 };
```

Ej en Memoria:

```
int valores[5] = { 1, 2, 3, 4, 5 };
```

```
printf("%p", valores);  
printf("%d", *valores);  
printf("%d", valores[0] );
```



Imprime la dirección donde comienza el vector. [0x1000]

Imprime el contenido de 0x1000. [1]

Imprime el contenido de la posición 0. [1]

Esto es porque en realidad valores[n] siendo $0 \leq n \leq 5$ es igual a $*(valores + n)$

Contenido	1	2	3	4	5
Dir. en Mem.	0x1000	0x1001	0x1002	0x1003	0x1004

Strings

En C no existe un tipo de dato String, en si, pero C nos provee una herramientas similar. En C los Strings nos son mas que arrays dinámicos o estáticos que siempre terminan en '\0' (carácter de fin de cadena).

Estáticamente:

```
char cadena[] = "Hola Mundo!";
```

```
cadena  => 0x100
```

```
*cadena => 'H'
```

```
&cadena => 0x100
```

Dinámicamente:

H	o	l	a	'	M	u	n	d	o	!	\0
0x100	0x101	0x102	0x103	0x104	0x105	0x106	0x107	0x108	0x109	0x10A	0x10B

```
char *cadena = malloc( strlen("Hola Mundo!") + 1 );
```


```
strcpy( cadena, "Hola Mundo!" );
```

```
cadena  => 0x200
```

```
&cadena => 0x1FF
```

```
*cadena => 'H'
```

0x1FF	0x200
-------	-------



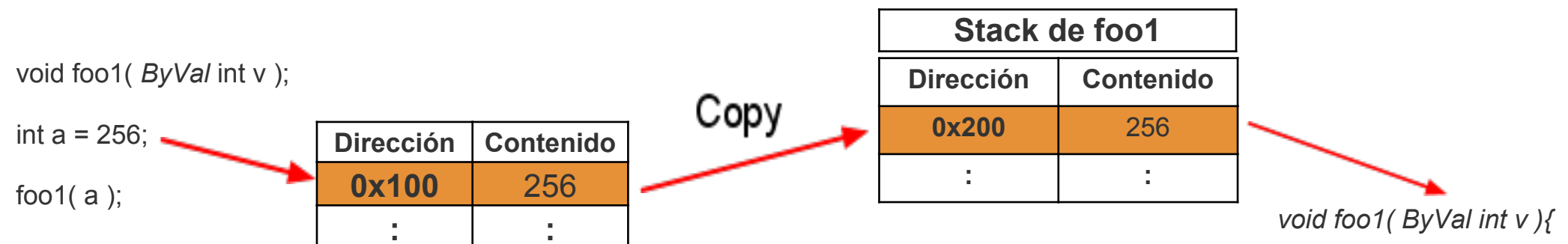
0x200	H
0x201	o
0x202	l
0x203	a
0x204	'
0x205	M
0x206	u
0x207	n
0x208	d
0x209	o
0x20A	!
0x20B	\0

Pasaje de Valores

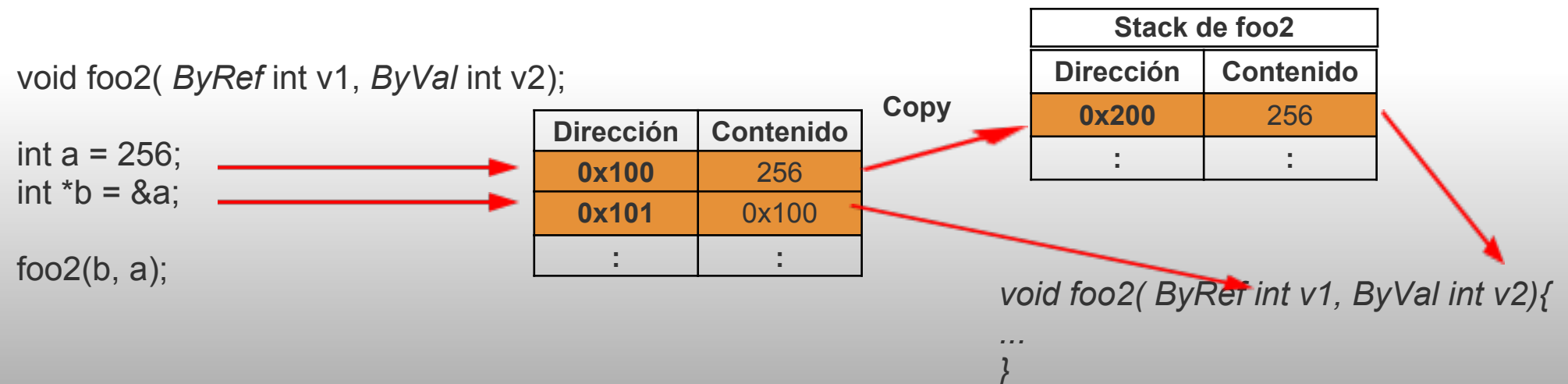
Formas de pasaje:

Cuando pasamos una variable a través de un argumento existen 2 mecanismos para realizar esto:

- **Por Valor** : Es cuando se pasa una copia de la variable, lo que sucede es que el valor de variable que estamos pasando es duplicada guardándola en el stack de la función.



- **Por Referencia** : Es cuando pasamos la dirección de la variable, con este método, la variable con la que trabaja el método es exactamente la misma que se le paso desde fuera del método



Formas de pasaje en C

En C no existe el pasaje por referencia, sino que todo es pasaje por valor. A fin de cuentas un puntero es una variable entera que contiene una dirección. Por lo que cuando se pasa un puntero a una función la variable que contiene la dirección (el puntero), se genera una copia en el stack de la función que contiene la misma dirección.

```
void foo2( int *v1 );
```

```
int a = 256;
```

```
int *b = &a;
```

```
foo2( b );
```

Dirección	Contenido
0x100	256
0x101	0x100
:	:

Copy

Stack de foo2	
Dirección	Contenido
0x200	0x100
:	:

Si nosotros dentro de foo2 hiciéramos:

```
v1 = 0xFFFF;
```

```
void foo2( int *v1){  
    ...  
}
```

La variable 'b' no se vería afectada, ya que v1 es una copia de 'b' y no 'b' en si. Pero si hacemos:

```
*v1 = 512;
```

Estamos modificando el contenido de la dirección que se encuentra en v1, es decir que estaríamos trabajando sobre 0x100 por lo que estaríamos cambiando el valor de 'b'.

Type Casting

- Consiste en cambiar de un tipo de dato a otro, para esto hay que considerar que en C todos son bytes y que los tipos solo definen rangos o cantidad de bytes al CPU. Solamente para el programador el tipo de dato puede tener una representación o visualización particular.
Es por esto que el casting, nos transforma los bytes sino que simplemente redefine la forma en la cual se van a leer e interpretar los datos siendo en todo caso aplicada una reducción de rango cuando se castea a un tipo de dato con diferente rango.

Ej1:
int a = 1000;
void *dir = (void*)a; /* Mismo rango */
char ch = (char)a; /* Diferente rango, perdida de bytes */

Ej2:
char *cadena = "Hola Mundo";
void val = (void*) cadena;

- El tipo void no requiere casteo explicito, es decir que la conversión de un tipo a otro no requiere casteo. El malloc es una función que devuelve void*, y es por eso que esta no requiere casteo:

```
char a = malloc( sizeof(char) );  
int b = malloc( sizeof( nt) );
```

- La conversión de enteros(int) a flotantes(float) o viceversa tienen un tratado especial por parte del compilador, ya que este, para estos casos, es suficientemente inteligente como para tratar de convertir de un valor a otro. Pero existe posibilidad de pérdida de datos o falta de precisión en ambas conversiones, lo cual hace que no sea recomendable hacerlo.

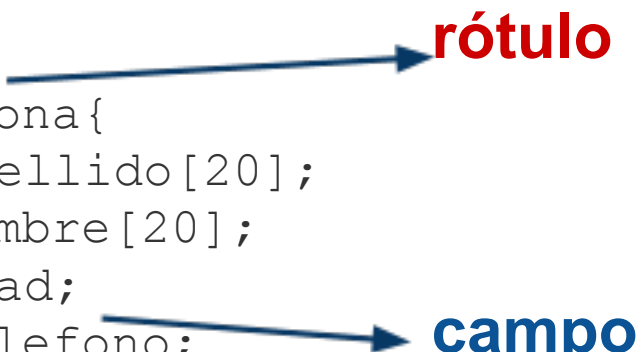
(From K&R, p141)

Un **struct** es una colección de una o más variables, de tipos posiblemente diferentes, agrupadas bajo ***un solo nombre*** para manejo conveniente.

Permiten organizar datos complicados, que implicarían más de una variable, y tratarlos a cada uno como una unidad en vez de por separado.

Ej:

```
struct Persona{  
    char apellido[20];  
    char nombre[20];  
    int  edad;  
    int  telefono;  
};
```



rótulo

campo

Como declarar una variable del tipo del struct (estáticamente)?

1) Declaro el struct antes, y defino la variable después:

```
struct Persona{  
char apellido[20];  
char nombre[20];  
char sexo;  
int edad;  
int telefono;  
};
```

```
#include <stdio.h>  
#include <stdlib.h>
```

```
int main(void){  
    struct Persona p;  
  
    strcpy(p.apellido, "Dumrauf");  
    p.edad = 22;  
    p.sexo = 'M';  
  
    printf("Apellido: %s\nEdad: %d\n", p.apellido, p.edad);  
  
    return EXIT_SUCCESS;  
}
```

2) Declaro el struct unicamente para indicar el tipo de la variable que defino :

```
#include <stdio.h>  
#include <stdlib.h>
```


```
int main(void){  
    struct{  
        char apellido[20];  
        char nombre[20];  
        char sexo;  
        int edad;  
        int telefono;  
    } p;
```

```
    strcpy(p.apellido, "Dumrauf");  
    p.edad = 22;  
    p.sexo = 'M';
```

```
    printf("Apellido: %s\nEdad: %d\n",  
           p.apellido, p.edad);
```

```
    return EXIT_SUCCESS;  
}
```

**El caracter '.' permite
acceder a un campo
de la estructura**



Anidar estructuras - structs como campos de otros structs

```
struct Documento{  
    char tipo[3];  
    int  numero;  
};
```

```
struct Persona{  
    char    apellido[20];  
    char    nombre[20];  
    char    sexo;  
    int     edad;  
    int     telefono;  
    struct Documento doc;  
};
```

Se accedería así:

```
struct Persona p;
```

```
strcpy(p.doc.tipo, "DNI");  
p.doc.numero = 33123456;
```

Definir una estructura dinámicamente

```
struct Persona *p = malloc(sizeof(Persona));
```

```
(*p).doc.numero = 33123456;  
strcpy((*p).doc.tipo, "DNI");
```

Al ser un puntero, primero
hay que acceder al
contenido del puntero, y
luego al campo

También se puede acceder de otra manera a los campos de un struct dinamico :

```
strcpy(p->apellido, "Dumrauf");
```

'->' indicaría "contenido de
la estructura referenciada
por el puntero"

Array de structs?? Pero por supuesto!

```
struct Persona familia[5];  
int i;  
  
for(i = 0; i < 5; i++)  
    strcpy(familia[i].apellido, "Dumrauf");  
  
strcpy(familia[0].nombre, "Matias");
```


(From K&R, p75)

Las **funciones** dividen tareas grandes en varias más pequeñas (**Modularizar**). Ocultan detalles de implementación (**Encapsulamiento**), dan claridad a la hora de leer código, facilitan la **Re-Utilización** del mismo y permiten un **fácil mantenimiento**.

Se distinguen 4 partes de una Función:

- Nombre
- Argumentos
- Retorno
- Cuerpo (encerrado por llaves, que indican un bloque). El cuerpo se compone de una lista de expresiones.

```
void saludarA(char *ente) {  
    printf("Hola %s\n", ente);  
};
```

Cuatro conceptos claves

.

LLAMADA : invocar la función.

DEFINICIÓN: la implementación.

```
void saludarA(char *ente) {  
    printf("Hola %s\n", ente);  
};
```

DECLARACIÓN: el prototipo de la función. Permite indicar que argumentos recibe y que valor retorna, sin necesidad de implementarla.

```
void saludarA(char *ente);    ó    void saludarA(char *);
```

SCOPE (Alcance): desde dónde y hasta dónde la función se puede Llamar. Ésto depende de en qué lugar del código la función fue Declarada ó Definida.

Una **LLAMADA** a función sólo se puede efectuar si la misma fue **DEFINIDA** ó **DECLARADA** en un **SCOPE** (Alcance) superior.

Ejemplo 1: Definiéndola arriba del main.

```
#include <stdio.h>
#include <stdlib.h>

void saludarA(char *ente) {
    printf("Hola %s\n", ente);
}

int main(void) {
    saludarA("Mundo");
    return EXIT_SUCCESS;
}
```

**DEFINICIÓN
de la Función**

**LLAMADA
a Función**

Ejemplo 2: **Declarándola arriba del main, pero definiéndola debajo.**

```
#include <stdio.h>
#include <stdlib.h>
```

```
void saludarA(char *ente);
int main(void) {
    saludarA("Mundo");
    return EXIT_SUCCESS;
}

void saludarA(char *ente) {
    printf("Hola %s\n", ente);
}
```

**DECLARACIÓN
de la Función**

**LLAMADA
a Función**

**DEFINICION
de la Función**

Ejemplo 3: Y la Declaración dónde está?

Ésto funciona? Está bien? Ambas? Ninguna de las dos?

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    saludaA( "Mundo" ); -----> Chan!!
    return EXIT_SUCCESS;
}

void saludaA(char *ente) {
    printf("Hola %s\n", ente);
}
```

Scopes

- Todo tiene un scope: variables, funciones, bloques, structs.
- Como C es un Lenguaje Estructurado y Procedural, el Scope rige de arriba hacia abajo dentro de un mismo archivo.

```
#include <stdio.h>
#include <stdlib.h>

void saludarA(char *ente);

int variableGlobal; -----> Scope Global al Main

int main(void) {
    char otraVariable; -----> Scope Global al Bloque Interno, pero
                                es Local al bloque del Main

    /* Bloque Interno */
    struct Persona *p; -----> Scope Global a todas las sentencias
                                dentro del bloque. Pero es local del bloque.
    puts("Imprimo en el Bloque Interno. Jojoojo.");
}

return EXIT_SUCCESS;
}
```

Cómo *ampliar* el Scope de una variable a varios source files?

Utilizando una declaración **extern**:

```
main_hello_world.c persona.c perro.c
1#include <stdio.h>
2#include <stdlib.h>
3
4void persona_saludarA(char *ente);
5void perro_saludarA(char *ente);
6
7int variableGlobal;
8
9int main(int argc, char **argv) {
10    variableGlobal = 1;
11
12    persona_saludarA("Mundo");
13    putchar('\n');
14    perro_saludarA("Mundo");
15    return EXIT_SUCCESS;
16}
```

```
Console
<terminated> c-talks [C/C++ Application]
Existen 1 Mundo/s
Hola Mundo!

Guau! Existen 1 Mundo/s
Guau Guau Mundo!
```

```
main_hello_world.c persona.c perro.c
1#include <stdio.h>
2
3extern int variableGlobal;
4
5void persona_saludarA(char *ente){
6    printf("Existen %d %s/s\n",
7        variableGlobal, ente);
8    printf("Hola %s!\n", ente);
9}
```

```
main_hello_world.c persona.c perro.c
1#include <stdio.h>
2
3extern int variableGlobal;
4
5void perro_saludarA(char *ente){
6    printf("Guau! Existen %d %s/s\n",
7        variableGlobal, ente);
8    printf("Guau Guau %s!\n", ente);
9}
```

OJO: Tener cuidado! No es recomendable usar variables globales (al Proceso/Programa) para todo.

Cómo *reducir* el Scope de una variable ó función para que sea privada de un único source file?

Utilizando una declaración **static**:

```
main_hello_world.c persona.c p
1#include <stdio.h>
2#include <stdlib.h>
3
4void persona_saludarA(char *ente);
5void perro_saludarA(char *ente);
6
7static int variableGlobal;
8
9int main(int argc, char **argv) {
10    variableGlobal = 1;
11
12    persona_saludarA("Mundo");
13    putchar('\n');
14    perro_saludarA("Mundo");
15    return EXIT_SUCCESS;
16}
```

```
main_hello_world.c persona.c perro.c
1#include <stdio.h>
2
3extern int variableGlobal;
4
5static void persona_saludarA(char *ente){
6    printf("Existen %d %s/s\n", variableGlobal, ente);
7    printf("Hola %s!\n", ente);
8}
9
```

Problems	
4 errors, 1 warning, 0 others	
Description	Resource
Errors (4 items)	
make: *** [c-talks] Error 1	c-talks
undefined reference to `persona_saludarA'	main_hello_world.c
undefined reference to `variableGlobal'	perro.c
undefined reference to `variableGlobal'	persona.c
Warnings (1 item)	
'persona_saludarA' defined but not used	persona.c

undefined reference to `persona_saludarA'

TAD

Tipo Abstracto de Dato

(From UTN-FRBA SSL Modulo IV K1GT5 - pag. 2)

Un **Tipo de Dato** es un conjunto de **valores** más un conjunto de **operaciones** aplicables sobre estos valores. Se clasifican en:

- **Tipos de Datos Primitivos:** provistos por el lenguaje.
Ej: **int** y sus operaciones son: +, -, *, /, %, <, >, !=, etc.
- **Tipos de Datos Abstractos (TADs):** creados por el programador (estructura y operaciones).
Ej: Pila, Cola, Lista, Persona.
Y operaciones como: meter y sacar, agregar y quitar, insertar y suprimir, saludarA.
- **Tipos de Datos Semi-Abstractos:** el lenguaje brinda algunas herramientas para construir el Tipo, y las operaciones en general las debe implementar el programador.
Ej: strings. Y operaciones como: strcmp, strcpy, strcat.

Ahh! ¿¡Eso era un TAD!?

Un TAD permite abstraernos de la implementación de los datos. La idea básicamente es **separar el uso del Tipo de Datos de su Implementación**.

A ésto se lo conoce como **Abstracción de Datos**.

Se centra en la **Modularización** y el **Encapsulamiento**.

Permite la re-utilización de código y el diseño e implementación de Bibliotecas propias.

Hay 3 pasos a seguir para crear un TAD:

1. Especificación
2. Diseño
3. Implementación

1. Especificación

Se debe analizar bien el Tipo de Dato que se quiere manipular.

TAD = (Valores, Operaciones)

Idealmente, uno debería poder especificar los valores y las operaciones ***sin tener que dar detalles de la implementación***, logrando una Abstracción completa.

Ésto se logra mediante el uso de Gramáticas, que permiten definir el/los alfabetos a partir de los cuales cada campo de un valor quedaría conformado. No vamos a meternos en tal nivel de detalle, sino que hablaremos a nivel implementación.

Preguntarse:

- Cuáles son los *valores* del Dominio (matemáticamente hablando)?
- Cuáles son las *operaciones*? Indicando cantidad y tipos de los operandos, y el valor de retorno. Es decir, el **Dominio** y la **Imágen** de cada operación.

REFERENCIAS: Enunciados, TPs y parciales de las cursadas de 2007.1C y 2008.1C de Sintaxis y Semánticas de los Lenguajes.
Prof. Ing. José María Sola. (Ver 'Bibliografía')

1. Especificación

Ej:

$$\text{TAD}_{\text{Persona}} = (\text{Persona}, \text{OP}_{\text{Persona}})$$

$\text{Persona} = \{ P \mid P = (\text{Apellido}, \text{Nombre}, \text{Sexo}, \text{Edad})$

y

Apellido	: Cadena
Nombre	: Cadena
Sexo	: char
Edad	: int

}

$\text{OP}_{\text{Persona}} = \{ \text{Persona_crear}, \text{Persona_matar}, \text{Persona_equals}, \text{Persona_saludarA},$
 $\text{Persona_cumplirAnos}, \text{Persona_esJubilado},$
 $\text{Persona_maldecir}, \text{Persona_falsificarIdentidad} \}$

$\text{Persona_crear} :: \text{Cadena} \times \text{Cadena} \times \text{Char} \times \text{int} \rightarrow \text{Persona}^*$

$\text{Persona_matar} :: \text{Persona}^* \rightarrow \text{Nada}$

$\text{Persona_equals} :: \text{Persona}^* \times \text{Persona}^* \rightarrow \text{int}$

$\text{Persona_saludarA} :: \text{Persona}^* \rightarrow \text{Nada}$

$\text{Persona_cumplirAños} :: \text{Persona}^* \rightarrow \text{Nada}$

$\text{Persona_esJubilado} :: \text{Persona}^* \rightarrow \text{int}$

$\text{Persona_maldecir} :: \text{Nada} \rightarrow \text{Nada}$

$\text{Persona_falsificarIdentidad} :: \text{Persona}^* \times \text{Cadena} \times \text{Cadena} \rightarrow \text{Persona}^*$

2. Diseño

En esta etapa se crea la **Interfaz** del TAD: el **header** (.h) (archivo encabezado).

La Interfaz es la **Parte Pública** de un TAD, la que posee los prototipos de las funciones y la declaración del TAD.

Con sólo ver la Interfaz uno puede **comprender** que hace ese TAD, y de ser necesario, se apoya en la documentación (si existe alguna).

3. Implementación

Finalmente, se comienza con la codificación de las funciones, es decir, la implementación del TAD. Ésto se hace en un **Source File**, un **.c**

La Implementación es la **Parte Privada** de un TAD, la que posee los **detalles** de lo que hace cada función.

Para el usuario del TAD ésto es transparente, ya que puede simplemente llamar a las funciones tantas veces como quiera, **abstrayéndose de la implementación** de las mismas.

Ejemplos en Eclipse.

3. Implementación

Para **una Interfaz**, pueden existir **muchas implementaciones** distintas.

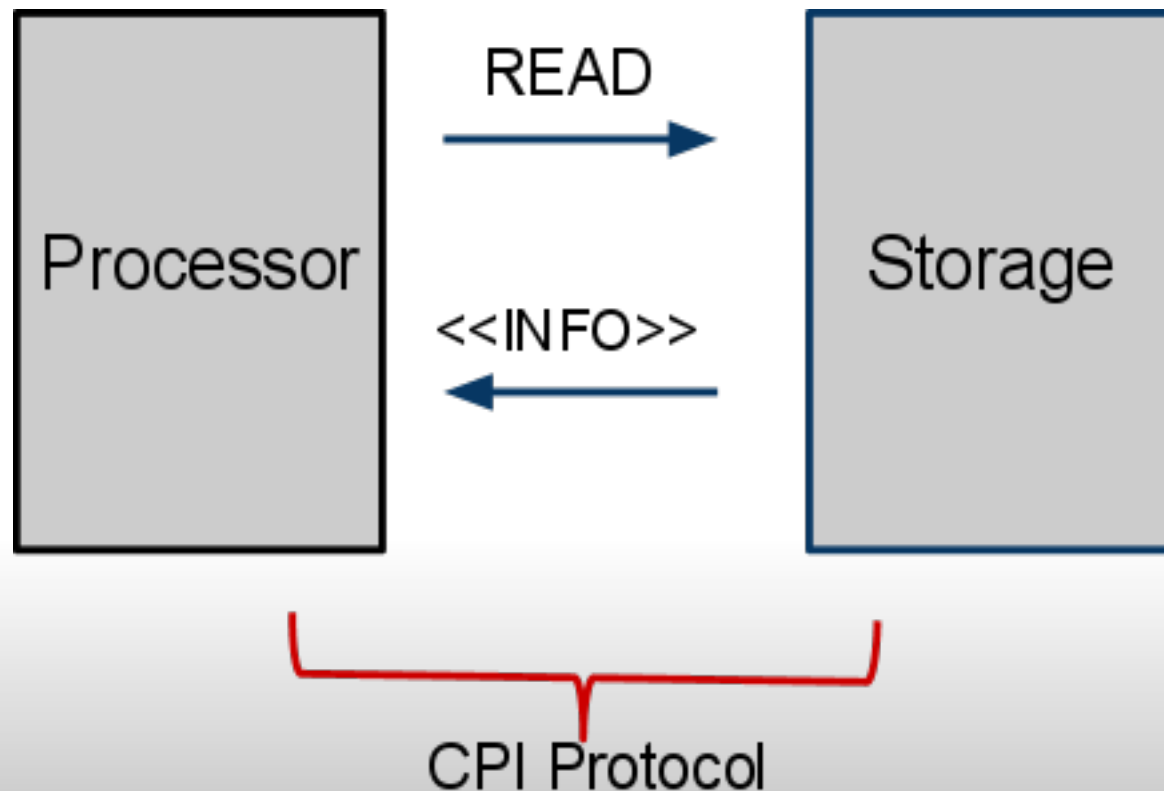
El hecho de trabajar con un header a nivel diseño, nos permite tener distribuir la Parte Pública de nuestro TAD, para que cada programador lo implemente a su manera.

Es tan simple como cambiar un Source File por otro.

Ejemplos en Eclipse.

Caso de Prueba I

Un proceso llamado "Processor", se comunica con un proceso "Storage" mediante un protocolo de comunicación red propio llamado CPI. El "Processor" envía paquetes cuya codificación indican si se quiere leer, escribir, borrar o actualizar.

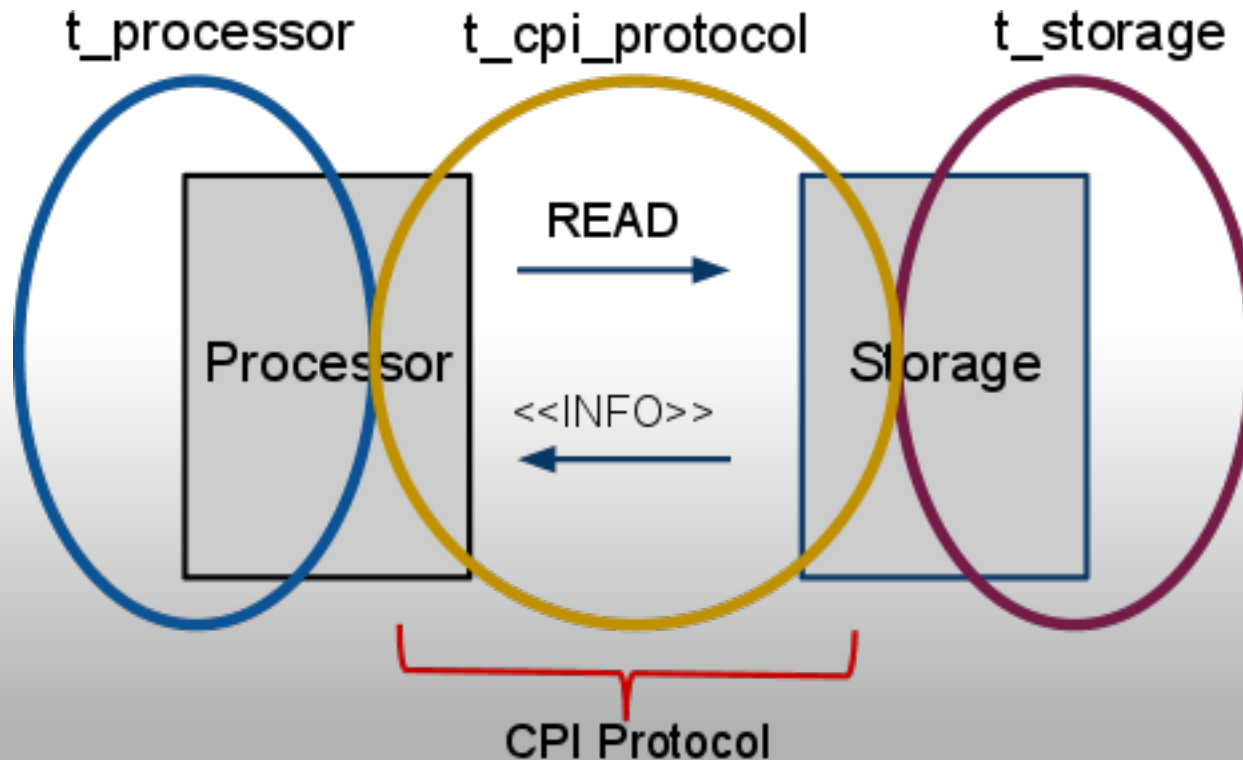


Inconvenientes:

- El Processor es desarrollado por 1 persona y el Storage por otra. Como logro coordinar el desarrollo el protocolo?
- Yo quiero asegurarme de que el protocolo funcione correctamente.
- El Processor resulta ser una tarea muy larga, mientras que el Storage es menor.
- El Storage al ser una tarea mas pequeña, va a terminar antes por lo cual necesita ser probada antes.
- Es posible que a medida que se va realizando el desarrollo del Processor, sea necesario probar algunas funcionalidades que requieren comunicación con el Storage. Comunicación que aun no se encuentra realizada.

Solución

Comprendemos que por cada proceso existe un TAD que lo representa (`t_processor` y `t_storage`). Pero consideramos al protocolo CPI como un TAD independiente (Es decir una librería que es utilizada por los otros 2 TADs). A esto lo podemos llamar interfaz, ya que actúa como una medio de comunicación entre los 2 TADs abstrayéndonos de la implementación.



Beneficios:

- Ya no tengo 2 tareas, sino tengo 3 y puedo delegar esta 3ra a alguien mas o a alguno de los que ya esta haciendo una de las otras tareas sin necesidad de tocar el código del otro.
- Es muy facil asegurar el funcionamiento del protocolo, porque ahora es una librería independiente la cual puedo testear.
- Tanto t_processor y t_storage incluyen t_cpi_protocol y utilizan sus funciones aunque estas no estén implementadas. Esto nos permite seguir codificando sin necesidad de esperar una implementación inmediata.
- En caso de necesitar realizar pruebas podemos definir nuestra propia implementación de t_cpi_protocol la cual simule respuestas.

```

.h cpi_protocol.h
*      Author: tacundo
*/

#ifndef CPI_PROTOCOL_H_
#define CPI_PROTOCOL_H_

typedef enum {
    CPI_READ           = 0x0,
    CPI_WRITE          = 0x21,
    CPI_DELETE         = 0x22,
    CPI_UPDATE         = 0x23
} e_cpi_pkg_type;

typedef enum {
    CPI_STORAGE        = 0x1,
    CPI_PROCESSOR      = 0x2
} e_cpi_mode;

typedef struct {
    int desc;
    struct sockaddr_in* my_addr;
} t_cpi_connection;

typedef struct {
    unsigned char      action;
    unsigned int       entry_id;
    void               *data;
} __attribute__((__packed__)) t_cpi_pkg;

t_cpi_connection  *cpi_protocol_create(e_cpi_mode mode, const char* ip, int port);
t_cpi_pkg         *cpi_protocol_read(t_cpi_connection *connection, unsigned int entry_id);
int               cpi_protocol_write(t_cpi_connection *connection, unsigned int entry_id, char data[]);
int               cpi_protocol_close(t_cpi_connection *connection);

#endif /* CPI_PROTOCOL_H_ */

```

[Esto no es un ejemplo completo]

Best Programming Practices

Best Programming Practices

- Variables con nombres significativos. **Basta de aux1, aux2, p1,ape, nom, etc.!**
- Nombres significativos para variables y funciones.
- Respetar una indentación de 4 espacios para sangría de un nuevo bloque.

Ej:

No

```
if( temperatura >= 20 ){  
  estufa_apagar();  
  persona_pasear();  
}
```

Sí!

```
if( temperatura >= 20 ){  
    estufa_apagar();  
    persona_pasear();  
}
```

- Utilizar TADs siempre que sea posible. Es muy útil a nivel Diseño de Procesos hacer un TAD por proceso, planteando bien la interfaz antes de saltar al código.
Ejemplos con Eclipse.

- Modularizar siempre que sea posible (y también necesario!)
- Seguir una nomenclatura para las funciones de cada TAD:
<tad>_<functionName>

Ej:

```
Persona_saludarA();  
Persona_comer();  
Persona_pasear();
```

Ejemplos con Eclipse.

Best Programming Practices

- No utilizar "Magic Numbers". Definirlos con MACROS representativas utilizando **define** y/o **enum** en donde sea posible, ya que facilita la lectura y también el refactoring del código.

`process_setPriority(process, 3); ¿¿??`

`#define PRIORITY_HIGH 3`

`Process_setPriority(process, PRIORITY_HIGH);`

`typedef enum {
 PRIORITY_HIGH = 3,
 PRIORITY_LOW = 1
}E_Priority;`

`Process_setPriority(process, PRIORITY_HIGH);`

- *Recomendación:* Kernighan & Ritchie Coding Style.

Estándar

```
int main(int argc, char *argv[])  
{  
    ...  
    while (x == y) {  
        something();  
        somethingelse();  
        if (some_error)  
            do_correct();  
        else  
            continue_as_usual();  
    }  
    finalthing();  
    ...  
}
```

Levemente Modificada

```
int main(int argc, char *argv[]){  
    ...  
    while (x == y){  
        something();  
        somethingelse();  
  
        if (some_error)  
            do_correct();  
        else  
            continue_as_usual();  
    }  
    finalthing();  
    ...  
}
```


Best Programming Practices

- Comentar el código sólo si es necesario, evitar comentarios redundantes. Un **buen código** "habla por sí mismo". Ej:

```
i++; // increment i
```

```
/**  
 * Vende un item  
 * @param item  
 * @return nada  
 */  
void iniciarVentaDelItem(t_item *item)
```

- Too Many Arguments

A medida que las funciones reciben mas argumentos se vuelven menos prolijas. En lineas generales mas de 3 argumentos ya se vuelve algo cuestionable, es posible que los datos no estén agrupados en un estructura o estén agrupados de forma poco conveniente.

- Output Arguments

Los argumentos son intuitivamente Input, utilizarlos como salida generan confusión o errores. Deben ser evitados, excepto cuando se evita el uso de memoria dinamica y se usa memoria estática intencionalmente.

```
char buff[100];  
file_read(file, buff);
```

```
char * buff = file_read(file);
```

Convencionamente se usa la palabra const delante de los argumento para indicar que no van a ser modificados.

```
void printf( const char* str, ... );
```

Best Programming Practices

- Tratar de evitar la anidación de **if**, dividir en subfunciones.
- Evitar el uso de **goto**
- Muchos programadores siguen las reglas de Edsger Dijkstra. Dijkstra decía que toda función tiene un punto de entrada y un punto de salida. Pero muchas veces el código es mas legible y simple si utilizamos múltiples puntos de salida ya sea usando `continue`, `break` o `return`.

```
void Persona_printName (t_persona *persona ){
    if(persona != null )
        return;
    if(persona->name != NULL )
        return;
    printf("%s", persona->name);
}
```

- Evitar el uso de variables globales, en un entorno multi-thread pueden aparecer varios errores de concurrencia.
- Definir structs utilizando **typedef** para "crear" un nuevo tipo de dato (un sinónimo).
- Definir/Declarar las variables ***al inicio de un bloque***, para que estén a la vista.

Best Programming Practices

- **Functions Should Do One Thing**

El nombre de la función describe lo que hace la función, es de buena práctica que 1 función haga solo 1 tarea. Esto puede generar funciones muy largas o que internamente la función haga cosas que el nombre de esta no describe por lo que quien la use no está al tanto de la funcionalidad. Ej:

```
t_list *paquetes = recibeYFiltraPaquetes(conexión);
```

debería ser

```
t_list *paquetes = recibePaquetes(conexión);  
paquetes = filtraPaquetes(paquetes);
```

- **Convenciones!**

Cuando se trabaja en equipo el principal problema es que todos vienen de aprendizajes o experiencias diferentes. Si no se fijan convenciones de código y trabajo al comienzo es muy posible que después sea complicado trabajar en el código de otro.

Bibliografía utilizada

- The C Programming Language 2nd Edition - Kernighan & Ritchie
- Rangos de tipos de datos: http://www.zator.com/Cpp/E2_2_4.htm
- Linux Programming Language Unleashed - Kurtwall
- Clean Code: A Handbook of Agile Software Craftsmanship - Robert C. Martin
- Tipos Abstractos de Datos: Módulo IV K1GT5 - Cátedra SSL (Muchnik-Sola)
- Notes on Coding Style for C Programming (from K&R Style): <http://www.cas.mcmaster.ca/~carette/SE3M04/2004/slides/CCodingStyle.html>
- TAD: Clases, TPs, enunciados y parciales de cursadas 2007.1C y 2008.1C de Sintaxis y Semánticas de los Lenguajes, UTN-FRBA. Prof. Ing. José María Sola. <http://groups.yahoo.com/group/UTNFRBASSL/> y Fotocopiadora UTN-FRBA.