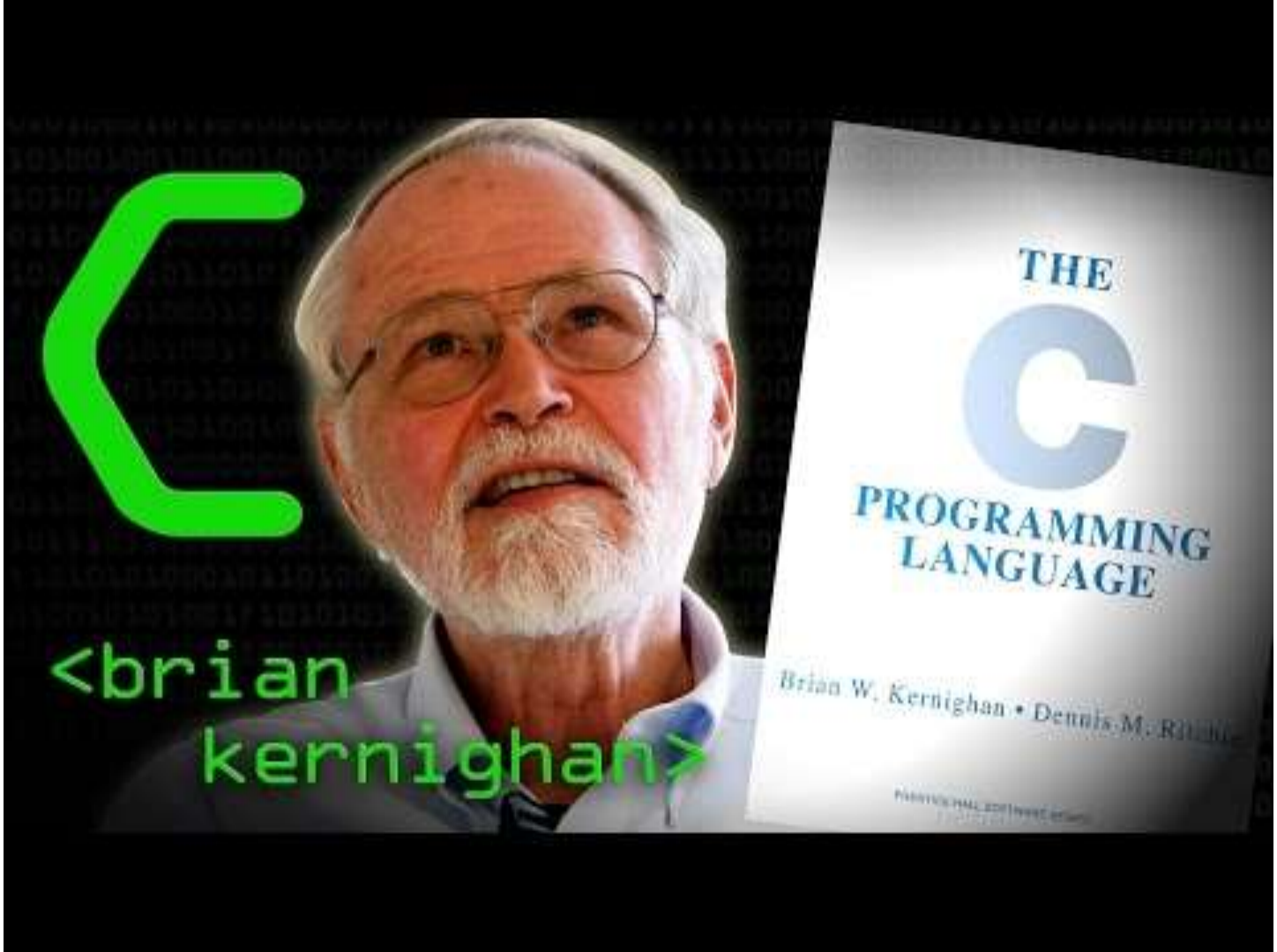
PROGRAMACIÓN DE SISTEMAS

El lenguaje de Programación C

Mg. Edith Giovanna Cano Mamani

El lenguaje de programación C

Basado en el curso: Effective Programming in C and UNIX del CMU


C
<brian
kernighan>
THE
C
PROGRAMMING
LANGUAGE
Brian W. Kernighan • Dennis M. Ri

Objetivos

- Entender la Programación de Alto y Bajo Nivel.
- Historia del Lenguaje de Programación C.
- Ejercicios en Lenguaje de Programación C.

Lenguajes de Bajo y Alto Nivel

Lenguaje de Bajo Nivel.

- Sus instrucciones consisten en ejercer un control directo sobre el hardware, se condiciona por la estructura física de la computadora.
- No implica que sea menos potente que uno Lenguaje de Alto Nivel.
- Usado en Sistemas Operativos o en controladores de dispositivos.
- Ejemplo. Lenguaje Ensamblador.

Lenguajes de Bajo y Alto Nivel

Lenguaje de Alto Nivel.

- Permite una mayor flexibilidad al abstraerse o ser literal. Permite una mejor comunicación entre lenguaje oral y el lenguaje máquina, es decir entre la escritura del programa y su compilación, varios de ellos están orientados a objetos,
- Expresa el algoritmo de acuerdo a la capacidad cognitiva humana.
- Ejemplo: Java, PL/SQL, C#, Cobol.

Historia

- 1969 El Lenguaje B (Ken Thompson), desarrollo Inicial basado en el Lenguaje BLCP (Martin Richard).
- 1970 Dennis Ritchie da el nombre de Unix derivado de MULTICS.
- 1972 Se crea el Lenguaje C (Dennis Ritchie).

Historia (…)

- 1974 Unix hace su primera aparición.
- 1978 Brian Kernighan y Dennis Ritchie publican el libro El Lenguaje de Programación C.
- 1980 Bjarne Stroustrup desarrolla el Lenguaje de Programación C++.
- 1990 Rectificación Estándar ISO.
- Unix inicialmente desarrollado en Lenguaje ensamblador y posteriormente en Lenguaje C.

Dennis Ritchie, Ken Thompson, and Brian Kernighan

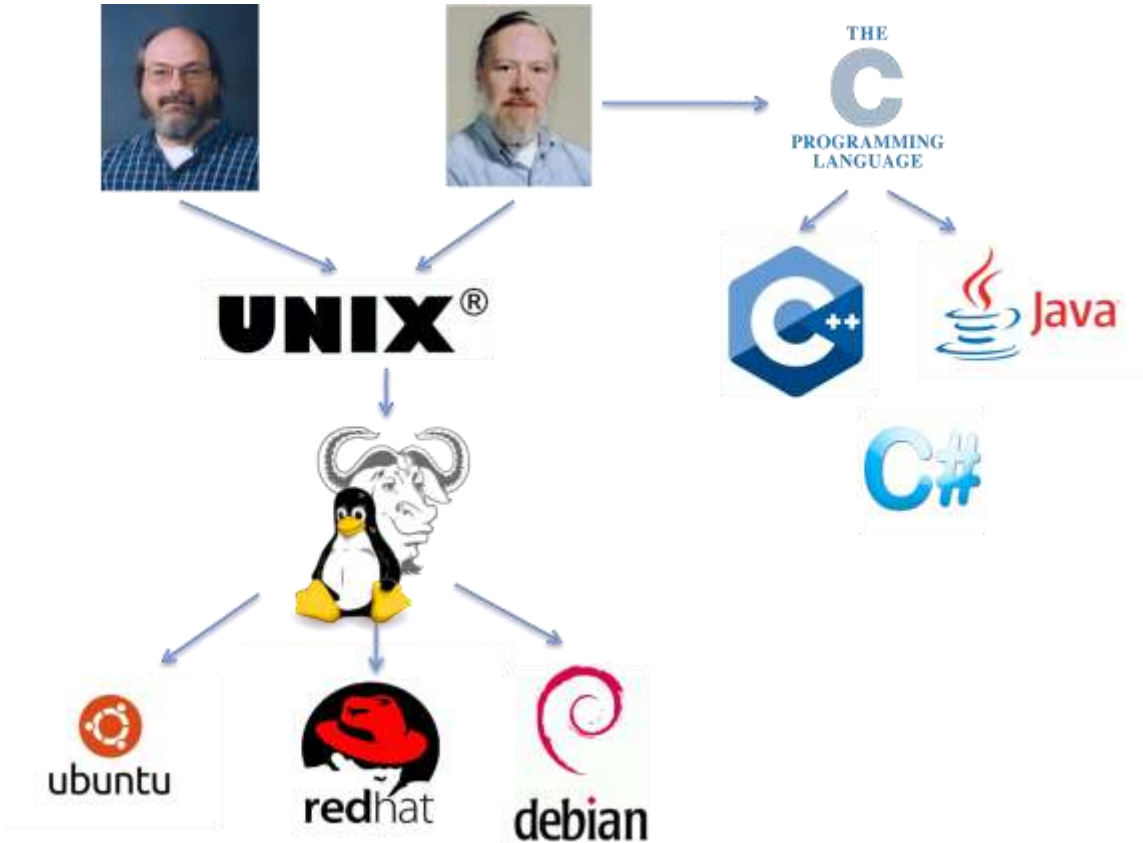
Características del Lenguaje C

- Utilización de Comandos Breves.
- Modelo de programación imperativo.
- Aparición de Funciones.
- Utilización de Punteros.
- Permite una programación de bajo nivel (Lenguaje Intermedio).
- Utilización de Arreglos.

Ventajas del Lenguaje C

- Permite que el programador pueda tener mucho mayor control y detalle sobre la utilización de la memoria por el programa (punteros).
  - Esto también podría verse como una desventaja, por que delega mayores responsabilidades al programador.
- Es un lenguaje de alto nivel.
- El código fuente es portable, aunque las bibliotecas siempre serán nativas del S.O.
- Genera código eficiente.

Derivados del Lenguaje C



Items para revisar en código

- Sintaxis
- Compilación
- Ejecución
- El Operador Sizeof
- Tipos de datos básicos
- Cualificadores
- Constantes entera y reales
- Constantes hexadeciamles
- Constantes Caracter
- Struct

Sintaxis básica

Un programa ejecutable en C requiere de una función denominada main, con el código a ser ejecutado por un programa. En el siguiente ejemplo se escribe un código que deberá ser escrito en un archivo con extensión .c (primero.c)

```
#include <stdio.h>

main(){

  printf(“Hello world\n”);

}
```

Compilación

Para ejecutar este código es necesario contar con un compilador que traduzca este código a algo que el sistema operativo pueda ejecutar. Nosotros usaremos gcc un compilador de software libre que entiende el lenguaje C (y también otros lenguajes)

```
$ gcc primero.c
```

Esto creará un archivo ejecutable denominado a.out, si se desea cambiar el nombre del archivo ejecutable generado se debe usar la opción -o, e indicar el nombre del archivo ejecutable que se desea.

Ejecución

Para ejecutar el programa se debe escribir

```
$ ./a.out
```

El punto slash (./) se usa para indicar que el shell debe buscar el programa ejecutable en el directorio actual y no las rutas señaladas por la variable PATH.

Cuando la compilación involucra varios archivos, es mejor usar una herramienta que automatice este proceso. Tradicionalmente se usó el programa make, que se verá más adelante.

Tipos de datos

Al igual que Java el lenguaje C es fuertemente tipado y posee una gran variedad de tipos y cualificadores, el tamaño de estos tipos será muy importante para nuestro curso, por lo que los describiremos indicando su tamaño. Es importante resaltar que el tamaño puede variar dependiendo del sistema operativo en el que se esté trabajando.

El Operador Sizeof

El operador unario sizeof() se utiliza para obtener el tamaño de un tipo de datos en bytes en bytes. No devolverá el tamaño de las variables o instancias. Al ser un operador hay que considerar la precedencia de operadores en su uso.

Revisar man de operadores:

```
$ man operator
```

Ejemplo:

Input : sizeof(byte);

Output : 1

Input : sizeof(int);

Output : 4

Tipos de datos básicos

Esto depende de la arquitectura del computador que se esté usando

Tipo	Tam. Bits	Dígitos de precisión	Rango	
			Min	Max
Bool	8	0	0	1
Char	8	2	-128	127
Signed char	8	2	-128	127
unsigned char	8	2	0	255
short int	16	4	-32,768	32,767
unsigned short int	16	4	0	65,535
Int	32	9	-2,147,483,648	2,147,483,647
unsigned int	32	9	0	4,294,967,295
long int	32	9	-2,147,483,648	2,147,483,647
unsigned long int	32	9	0	4,294,967,295
long long int	64	18	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long int	64	18	0	18,446,744,073,709,551,615
Float	32	6	1.17549e-38	3.40282e+38
Double	64	15	2.22507e-308	1.79769e+308

Cualificadores

Los calificadores de tipo son las palabras clave que describen semánticas adicionales sobre un tipo. Son una parte integral de las firmas de tipo.

Los calificadores de tipo tienen la misión de modificar el rango de valores de un determinado tipo de variable. Estos calificadores son cuatro:

signed

Le indica a la variable que va a llevar signo. Es el utilizado por defecto.

Tipo		Tamaño	Rangos de valores
signed char		1 byte	-128 a 127
signed int		2 bytes	-32768 a 32767

Cualificadores

unsigned

Le indica a la variable que no va a llevar signo (sin valores negativos).

Tipo	Tamaño	Rangos de valores
unsigned char	1 byte	0 a 255
unsigned int	2 bytes	0 a 65535

short

Rango de valores en formato corto (limitado). Es el utilizado por defecto.

Tipo	Tamaño	Rangos de valores
short int	2 bytes	-32768 a 32767

Cualificadores

long

Rango de valores en formato largo (ampliado).

Tipo	Tamaño	Rangos de valores
long int	4 byte	-2.147.483.648 a 2.147.483.647
long double	10 bytes	-3'36 E-4932 a 1'18 E+4932

También es posible combinar calificadores entre sí:

signed long int = long int = long

unsigned long int = unsigned long 4 bytes 0 a 4.294.967.295 (El mayor entero permitido en 'C')

Constantes Enteras

Cualquier número escrito es una constante, sin embargo hay tratamientos distintos para enteros, reales y caracteres.

En el caso de números enteros, se puede especificar si se requiere algo de memoria extra para almacenarlos agregando una l o L al final del número; por ejemplo, 65535L.

También se puede especificar si el número se almacenará sin signo agregando la letra U o u al final; se pueden combinar las letras ul o UL si se necesita.

Constantes reales

Para el caso de los números de punto flotante, el punto decimal define claramente su tipo de valor, sin embargo también se pueden escribir usando la notación científica agregando la letra e como indicador del inicio del número del exponente; por ejemplo 5e-2 equivale a 0.05; agregando la letra l o L se puede especificar que la constante se debe almacenar como un long double

Constantes Hexadecimales y Octales

La representación de números hexadecimales y octales es importante porque permite una escritura abreviada de números binarios, que son los que realmente entiende el computador. Para representar números constantes octales es suficiente poner un 0 (cero) a la izquierda los dígitos.

Para representar números hexadecimales se debe agregar `0X` o `0x` a la izquierda de los dígitos.

Constantes Caracter

A diferencia de Java, en C la representación directa de un caracter es directamente como número, aunque su impresión sea de un caracter de texto. Al ser números, es posible sumarles valores o restarlos directamente, o compararlos como si fueran números, sin ningún tipo de conversión. Esto tiene un impacto importante en la concepción de programas que trabajen con Strings, que como se verá más adelante, son arreglos de caracteres.

Struct

struct ejemplo { char c; int i;}; La palabra reservada struct indica se está definiendo una estructura. El identificador ejemplo es el nombre de la estructura. Las variables declaradas dentro de las llaves de la definición de estructura son los miembros de la estructura.

struct ejemplo { char c; int i;};

Struct y clases

De manera similar a las clases de Java, en C es posible agrupar tipos de datos distintos en una sola entidad usando struct, sin embargo struct no crea un nuevo tipo de datos y tampoco permite asociar métodos, así que sólo se usa para **la agrupación de datos**.

```
struct Complejo {

  double real;

  doulbe img;

};
```

Es importante hacer notar, que estas variables estarán en regiones de memoria contigua, aunque esto no parece muy relevante en este momento, será una de las propiedades más importantes que ofrece el lenguaje C y que permite usar las estructuras de datos en la solución de problemas reales.

Variables de tipo Struct

La declaración de una variable del tipo de la estructura requiere que se use la palabra struct necesariamente

```
struct Complejo c1;
```

El acceso a los datos de una estructura es idéntico al acceso de los atributos de un objeto en java, es decir, usando el operador punto (.).

```
c1.real = 1.5;
```

```
c2.img = 3.2;
```

Las estructuras no cuentan con un constructor, pero también es posible inicializar una nueva estructura con valores iniciales.

```
struct Complejo c2 = {4.3, 5.7};
```

Funciones que devuelven struct

Es posible crear funciones que devuelvan nuevas estructuras inicializadas, al estilo de los constructores de java, pero estos no formarán parte de la estructura, serán funciones simples sin ninguna relación sintáctica con el tipo de dato struct.

```
struct Complejo creaComplejo(double real, double img){

  struct Complejo c;

  c.real = real;

  c.img = img;

  return c;

}
```

Estructuras de datos con Struct

Se puede usar la agrupación de datos que crea Struct para crear estructuras de datos como listas, por ejemplo

```
struct Node {

  int data;

  struct Node* next;

};

struct Node *list = null;
```

En este ejemplo el dato next, será un puntero al siguiente nodo, el tema de punteros se verá en la próxima clase.

typedef

Permite crear nuevos tipos de datos, por ejemplo podemos crear el siguiente nuevo tipo

```
typedef unsigned int size_t
```

En este caso size_t es un sinónimo para unsigned int.

```
size_t x = 2;
```

Debido a que un código de C puede ser ejecutado en distintas arquitecturas (máquinas) en algunas de ellas los tipos de datos pueden tener distinto tamaño, sin embargo size_t puede ser usado para focalizar este problema únicamente en la declaración de **`size_t`**.

Typedef y struct

La combinación de typedef y struct puede ser muy útil en la implementación de estructuras de datos, recordando el ejemplo anterior

```
typedef struct Node {
  int data;
  ListNode* next;
} ListNode;
ListNode *list = null;
```

¿Preguntas?

Procesos

https://tldp.org/LDP/intro-linux/html/sect_04_06.html

Entrada y salida

https://tldp.org/LDP/intro-linux/html/sect_05_05.html

PROGRAMACIÓN DE SISTEMAS

PUNTEROS

Mg. Edith Giovanna Cano Mamani

El lenguaje de Programación C - punteros

Programación de sistemas

Declaración de punteros

Quizá la herramienta más poderosa que ofrece el lenguaje C es el acceso a la memoria, a través de punteros; sin embargo, también es la mayor fuente de problemas y errores de programación, por lo que hay que ser muy cuidadosos con su uso.

Los punteros son variables que permiten almacenar direcciones de memoria de un tipo determinado, así por ejemplo podemos tener punteros a enteros, reales e incluso estructuras.

Para declarar una variable de tipo puntero, se debe agregar un asterisco:

```
int* pi;

float* pf;

double* pd;
```

En estas declaraciones, las variables no contienen ningún valor.

Inicialización de punteros

Se puede obtener la dirección de memoria de una variable usando el operador ampersand (&), las variables de tipo puntero están especialmente creadas para almacenar estas direcciones.

```
int i = 2;

int* pi = &i;
```

Ahora la variable pi apunta a la variable i. Para ser más precisos, i no es una variable, sino un **identificador** de la variable, la variable se encuentra en la memoria del computador almacenando el valor 2; entonces i sólo es el nombre que puede usar un programador para manipular la variable que está en la memoria.

Accediendo al valor apuntado

Con el puntero pi apuntando a i, se puede usar el operador asterísco (*) para acceder a la misma variable en memoria que i.

```
*pi = 3;

printf(“%d\n”, i); // imprimirá 3

(*pi)++;

printf(“%d\n”, i); // imprimirá 4
```

En este punto tanto i como *pi son sinónimos, son dos nombres para la misma variable, así que ambas se podrán usar indistintamente para hacer cualquier tipo de operación aritmética sobre enteros. En Java las variables que contienen objetos se comportan como si fueran punteros a variables, de hecho también se pueden crear sinónimos e incluso, cuando se imprime su valor, se puede ver algo similar a una dirección de memoria.

Declaración de la función swap

Puede que aún la utilidad de los punteros no sea muy clara, pero ya es posible que la utilicemos para algo que no se podría hacer en Java con tipos primitivos: la función de intercambio de valores swap

```
swap(int *pi = &i, int *pj = &j)
```

```
void swap(int *pi, int* pj){
  int tmp = *pi;
  *pi = *pj;
  *pj = tmp;
}
```

El objetivo de esta función es intercambiar el valor de dos variables

```
void foo(){
  int i = 2;
  int j = 3;
  swap(&i, &j);
  printf("i = %d; j =%d", i, j); // imprime i = 3; j = 2
}
```

Uso de la función swap

```
int i = 2;

int j = 3;

swap(&i, &j);

print(“%d, %d”, i, j); // imprimirá 3, 2
```

La función swap ha intercambiado el valor de dos variables de un contexto ajeno al suyo; La posibilidad de crear funciones que manipulen variables externas a su propio contexto, ciertamente puede ser peligrosa y hasta indeseable; en general el manejo de punteros es como un poder super poderoso y muy difícil de controlar.

Punteros – Recomendaciones de uso

- Operador &, usado para indicar la dirección de la variable.
- Operador *, usado para acceder al valor del puntero.
- Operador &, usado en la invocación de funciones, de tal manera que va delante de los parámetros de entrada y/o salida.
- En la definición de la función, cuando el parámetro de entrada y/o salida se debe definir al puntero anteponiendo *.

Tamaño de una variable puntero

Sea cual sea el tipo de dato al que se apunte, las variables de tipo puntero siempre tendrán el mismo tamaño, esto implica que las direcciones de memoria serán del mismo tamaño, sin importar el tamaño de dato al que se apunte, esto no hace que las variables de tipo puntero sean intercambiables libremente, pero si es posible tener una variable de tipo puntero genérica, que pueda apuntar a cualquier tipo.

```
int i = 2;

void* pi = &i;
```

En este caso, no será posible tratar a *pi como si fuera un entero cualquiera; sin embargo, esta posibilidad de apuntar a cualquier tipo de puntero deja la puerta abierta para el concepto de polimorfismo.

Arreglos

De manera similar a struct, los array también permiten crear conjuntos de datos de memoria contigua, pero en este caso todos los datos son del mismo tipo.

Para crear un arreglo se deben usar los corchetes, indicando el tamaño del arreglo.

```
int a[10];
```

Esto crea espacio para 10 enteros en memoria continua y para acceder a ellos se deben usar índices de manera similar a Java; sin embargo, en el lenguaje C, los arreglos no tienen valores iniciales, entonces se debe suponer que contendrán valores aleatorios, a los que también se les conoce como basura.

```
a[0] = 2;
a[1] = 13;
```

Arreglos de C vs Java

Otra diferencia con java es que los arreglos no son objetos, por lo que no hay manera de determinar su tamaño y si se intenta acceder a índices más allá de su tamaño, se obtendrán comportamientos muy extraños, que muchas veces son muy difíciles de depurar; la mejor técnica de programación en este entorno es usar constantes para el tamaño del arreglo.

Arreglos y punteros

Los arreglos y los punteros están muy relacionados en el lenguaje C (al igual que los struct). Por ejemplo, se puede crear un puntero al primer elemento de un arreglo de la siguiente manera:

```
int* pa = &a[0];
```

Después de esta línea tanto *pa, como a[0], son sinónimos y se refieren a la misma variable en memoria; incluso se puede acceder al resto del arreglo usando lo que se conoce como aritmética de punteros

```
*(pa + 1) ++;
```

Esta instrucción hará que el segundo elemento del arreglo (a[1]) se incremente en uno y valga 14. Con este comportamiento se puede entender con más claridad el motivo de que los índices de un arreglo empiecen en cero y no en uno (como en la matemática), los índices del arreglo son entonces desplazamientos.

Aritmética de punteros

Cuando se crea un puntero de un tipo dado (por ejemplo, puntero a entero) y se le suma o resta valores, hay que notar que estos valores no son tomados como cantidades fijas, sino más bien variables dependiendo del tipo de puntero con el que se está trabajando.

En la instrucción *(pa + 1)++, la operación pa + 1, no incrementa la dirección de memoria contenida en pa en uno, sino en el tamaño de un entero en bytes. Si pa estuviera apuntando a otro tipo de valor (como char, double, etc.) el incremento sería de tamaño distinto. En terminos generales el valor de 1 de la expresión pa + 1, representa el tamaño de un valor del tipo de puntero dado. En la aritmética de punteros, no sólo se puede usar el operador + para aumentar la dirección apuntada, sino también - para decrementar la misma dirección, bajo las mismas reglas.

Strings

Los strings en C, se tratan como si fueran arreglos de char con un valor de cero al final. Esto hace que sea mucho más fácil acceder a los caracteres del string usando simples operadores de arreglos, a diferencia de java que se debe usar métodos de la clase String.

```
char unsa[] = “Universidad Nacional de San Agustín de Arequipa”;
int i = 0;
while(unsa[i] != 0){
  i++;
}
```

En este código unsa es un arreglo de caracteres, donde cada posición contiene un el valor de un caracter y la última posición del arreglo contiene un valor cero, indicando el fin del string; este valor se agrega de manera automática. Por esto el ciclo while puede hacer la comparación con cero, sabiendo que el string finaliza con este valor. El valor de unsa[0] es igual al caracter A, mientras que el valor de unsa[1] es el valor n.

Modificación de Strings

El siguiente código puede modificar el string en una sóla línea.

```
char unsa[] = “Universidad Nacional de San Agustín de Arequipa”;

unsa[1] = ‘N’;
```

Ahora el string unsa será “UNiversidad Nacional de San Agustín de Arequipa”.
También es posible crear un string usando el operador asterisco.

```
char* aqp = “Arequipa”;
```

Todas las reglas anteriores se aplican a este string, sólo que no es posible modificar sus caracteres, no por el uso del tipo puntero, sino porque la zona de memoria a la que este apunta es de sólo lectura (zona del código).

Concatenación de Strings

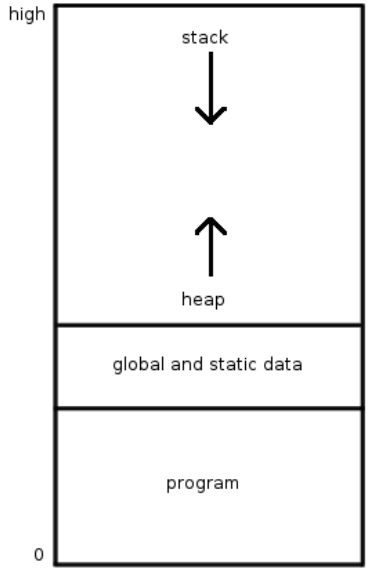
Aunque en C, la manipulación de strings se reduce a la simple manipulación de arreglos, la concatenación de arreglos puede ser una tarea más complicada, porque ahora se deberá aplicar también las mismas reglas de concatenación de arreglos.

```
#include <stdio.h>

char *strcat(char *, char *);
int main(int argc, char **argv){
          char str[256];
          str[0] = 0;
          strcat(str, "Escuela ");
          strcat(str, "Profesional de ");
          strcat(str, "Ingeniería de
Sistemas");
          printf("%s\n", str);
}
char *strcat(char *left, char *right){
          char *resp = left;
          while(*left) left++;
          while(*right){
                    *left = *right;
                    left++;
          right++;
          }
          *left = 0;
          return resp;
}
```

Regiones de memoria

Un programa en C tiene algunas regiones de memoria bien definidas:

- la sección de código, donde está el código (en bits) de nuestro programa y es de sólo lectura;
- la sección de la stack, donde se almacenan las variables y llamadas a funciones, y
- la sección del heap, donde el programador puede reservar y liberar memoria según la necesite.

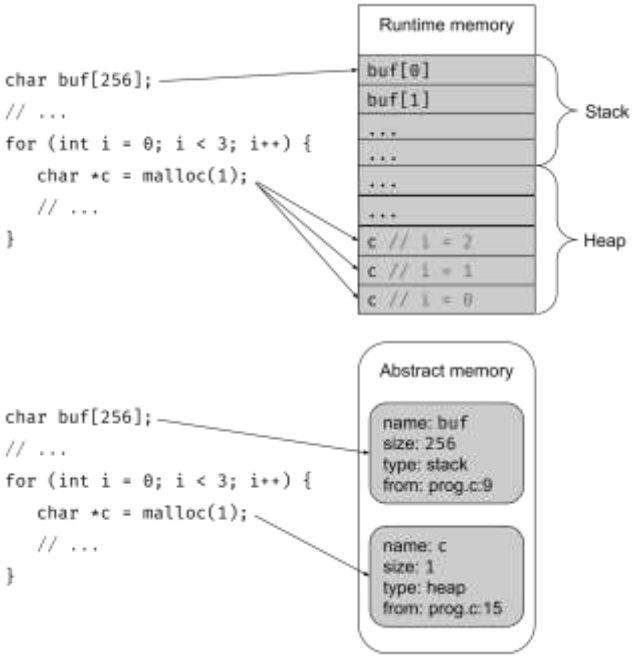


Punteros al stack

Con los punteros podemos apuntar a cualquier región de memoria e incluso modificarla, excepto la región de código, que es de sólo lectura. Un puntero a la stack se obtiene de manera muy natural:

```
int i = 2;

int* pi = &i;
```

El puntero pi está apuntando a la región de memoria del stack



Punteros – Memoria y Variables

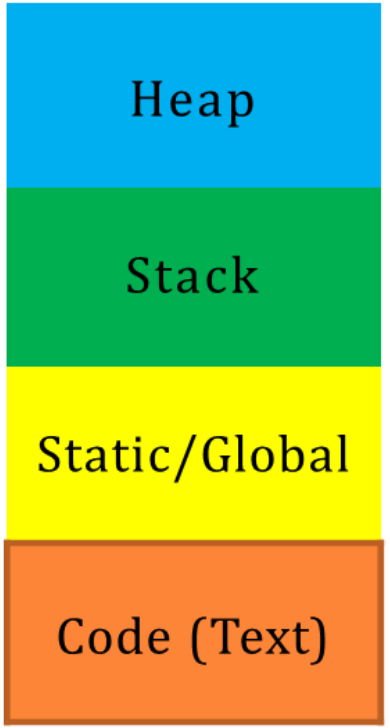
```c
#include <stdio.h>
int total;

int Cuadrado(int x){
   return x*x;
}

int SumaCuadrado(int x, int y)
{
   int z = Cuadrado(x+y);
   return z;
}

int main(){
   int a=4, b=8;
   total = SumaCuadrado(a,b);
   printf("Total = %d",total);
}
```

Almacenamiento libre

Llamadas a funciones y a variables locales

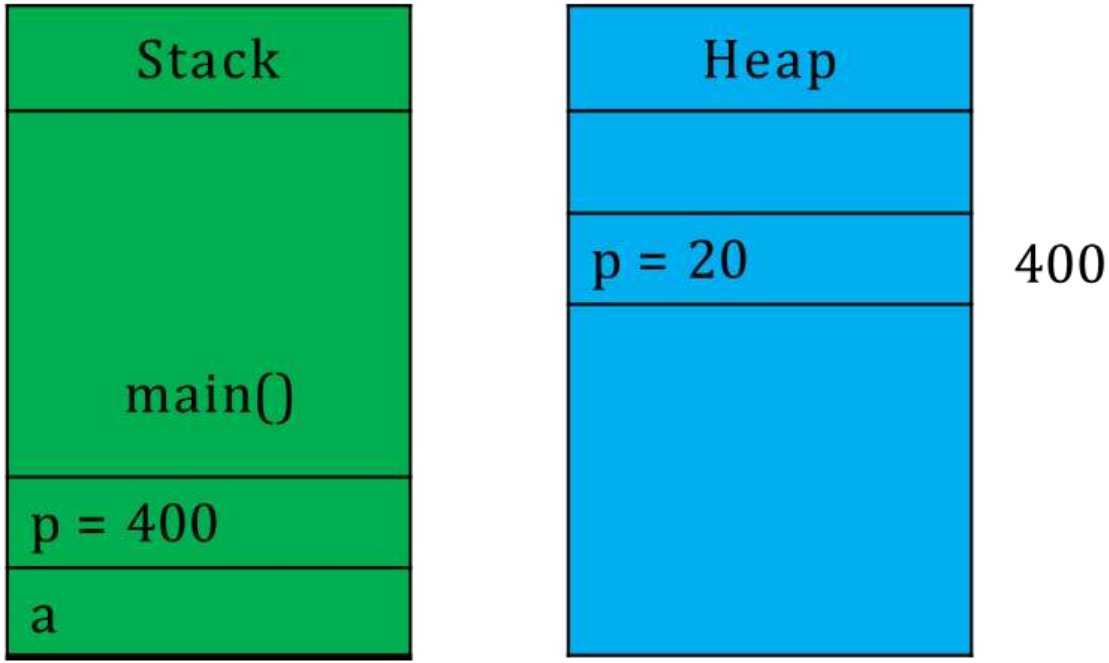
Global

Instrucciones



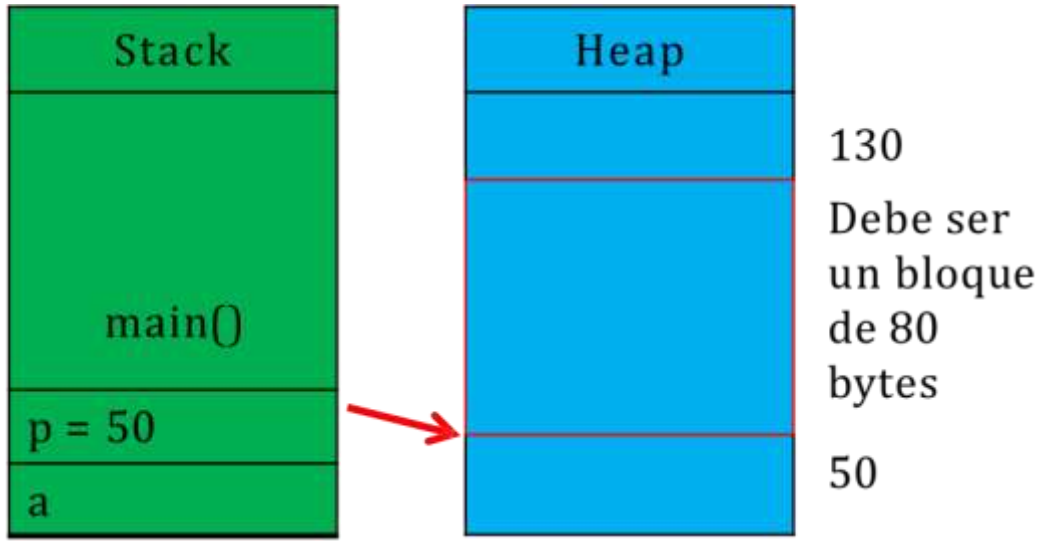
Punteros – Memoria y Variables

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

int main()
{
    int a;
    int *p;
    p = (int*) malloc(sizeof(int));
    *p = 10;
    p= (int*) malloc(sizeof(int));
    *p=20;
}
```



Punteros – Memoria y Variables

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

int main()
{
    int a;
    int *p;
    p = (int*) malloc(sizeof(int));
    *p = 10;
    p= (int*) malloc(20*sizeof(int));
    *p=20;
}
```



p[0], p[1], p[2]

*p, *(p+1), *(p+2)

Punteros – Memoria y Variables

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

int main()
{
    int a;
    int *p;
    p = (int*) malloc(sizeof(int));
    *p = 10;
    p= (int*) malloc(20*sizeof(int));
    *p=20;
}
```

Punteros a la sección de código

Un puntero a la región de código se puede obtener con la declaración de un string:

```
char* str = “Un mensaje”;
```

El puntero str, apunta a la región de memoria del código.

Asignación dinámica de memoria

Para conseguir que un puntero apunte a la sección del heap se debe usar la función void* alloc(n), que recibe el número de bytes que se desean reservar y devuelve la dirección del primero de estos. Al devolver void*, el puntero no tiene un tipo determinado, así que todo depende del tipo de puntero donde se almacene esa dirección.

```
int *pi = malloc(sizeof(int) * 10);
```

En este código usa el operador sizeof para saber el tamaño de un entero en bytes y luego se multiplica este valor por 10, con lo que se tendrá espacio para 10 enteros consecutivos, siendo el primero de ellos *(pi + 0) y el último *(pi + 9).

Liberación de memoria dinámica

A diferencia de la sección de stack, la memoria de la sección del heap debe ser liberada de manera explícita usando la función free(void*) que desasigna la memoria que se había reservado para una variable.

```
int *pi = malloc(sizeof(int) * 10);

…

free(pi);
```

Si no se desasigna la memoria, ésta permanecerá asignada permanentemente, mientras el programa esté en ejecución, pudiendo en algún momento consumir toda la memoria del computador y provocando una falla general. Tanto malloc, como free son funciones de la biblioteca <stdlib.h>

JERARQUÍA DE MEMORIA

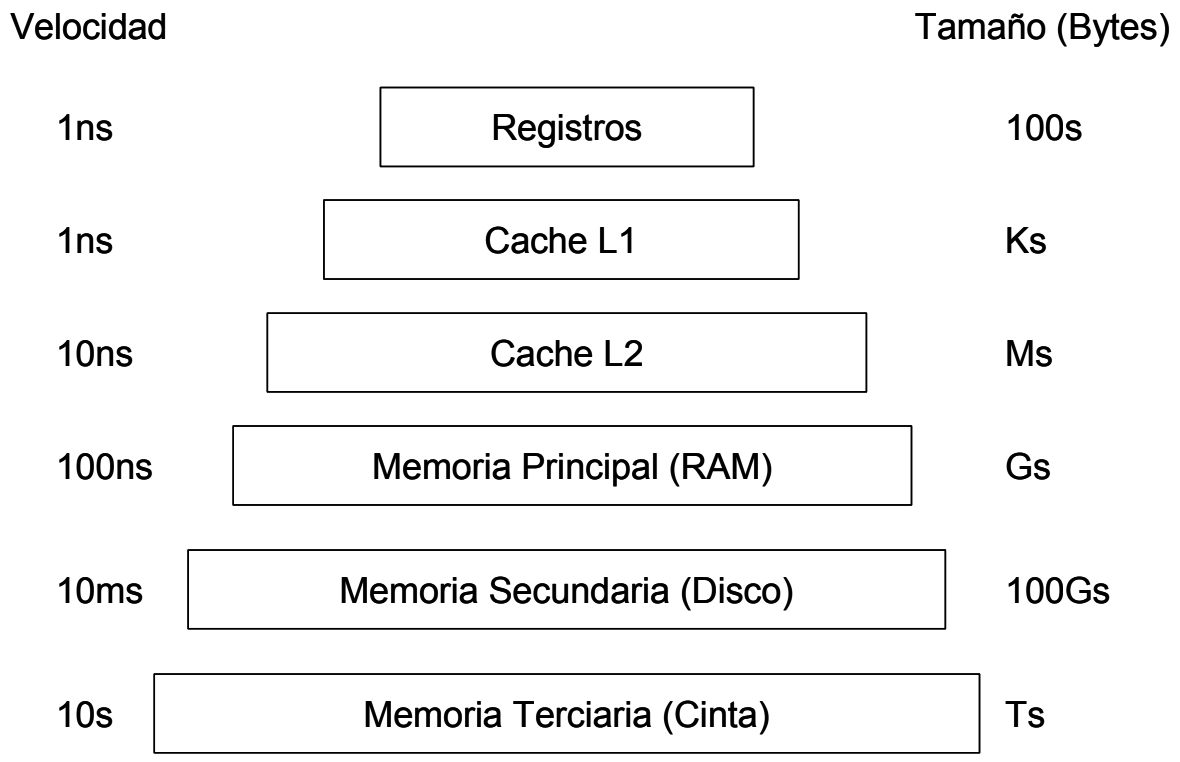
1 Introducción

Este capítulo está dedicado al análisis de los distintos niveles de memoria existentes en un computador, con especial énfasis en los sistemas de "cache".

2 Justificación Tecnológica

Históricamente siempre existió un compromiso de diseño en los sistemas de memoria: las memorias rápidas son más costosas que las memorias más lentas. Por tanto un diseño equilibrado es aquel que combina en su justa medida memorias de un tipo y de otro. Este diseño debe, además, lograr la mejor relación entre el coste del sistema y su rendimiento (capacidad de proceso).

La solución a este problema es el concepto de *jerarquía de memoria*. Esta jerarquía básicamente establece un orden en función de la capacidad de memoria y su velocidad. La mayor jerarquía la tiene la memoria más rápida (y por tanto menor en tamaño, para mantener el coste acotado) y la menor la memoria más lenta (y por tanto más económica) pero por tanto más abundante.

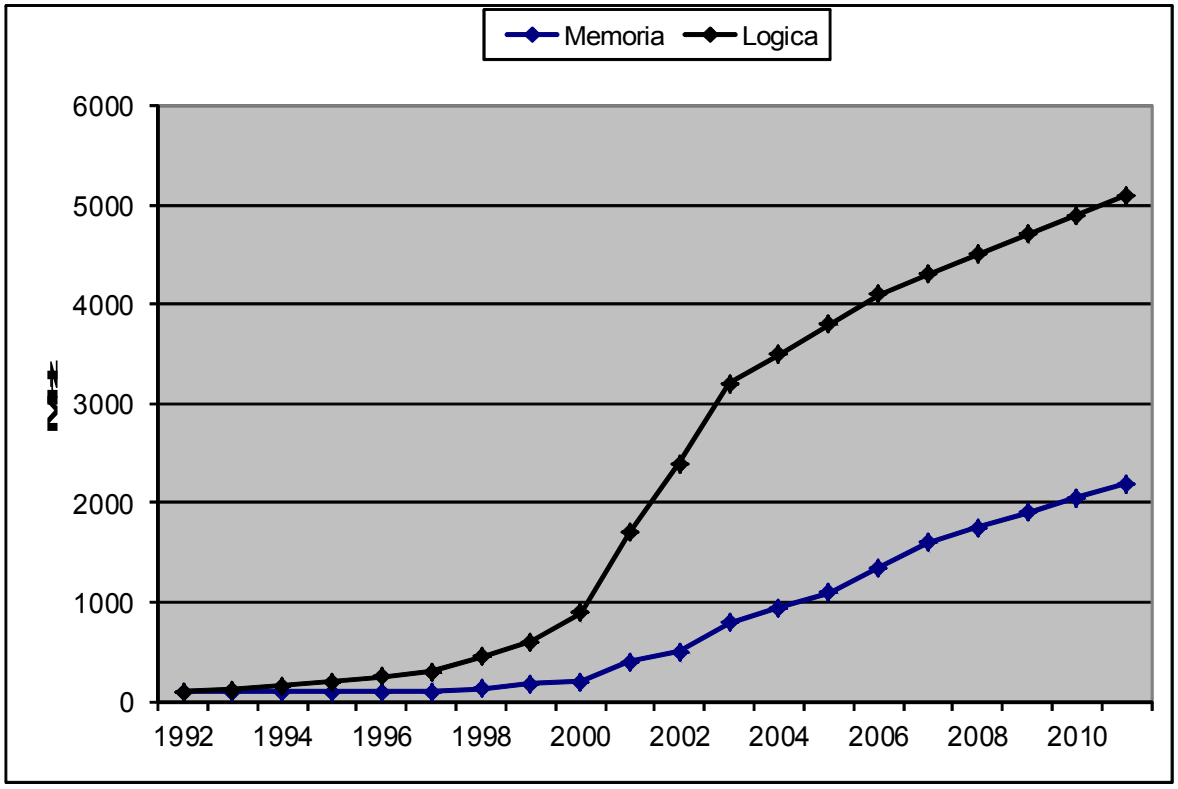


Cuanto más descendemos en la jerarquía de memoria, más económica (y por tanto más abundante) es la memoria. Por ejemplo en la actualidad una memoria RAM de 4 GBytes cuesta unos U$S 40, es decir unos 10 U$S / GByte. Un disco de 500 GBytes cuesta del orden de U$S 100, es decir unos 0,2 U$S / GByte. Aquí vemos que en materia de precio por capacidad de almacenamiento el disco le lleva una ventaja de 50 a 1 a la memoria RAM. Si lo miramos velocidad de acceso el disco es 100000 veces más lento.

Por su lado un chip de memoria SRAM (construido con tecnología de flip-flops) de 4

Mb cuesta unos U$S 8, mientras que un chip DRAM (construido con tecnología de memorias dinámicas) de 2 Gb sale U$S 1. Esto significa que la memoria SRAM cuesta 4000 veces más que la memoria DRAM !!. Esto lleva a la inviabilidad económica de construir computadoras con memoria principal basada en tecnología SRAM.

Por otra parte en la década de los 90 del siglo pasado las tecnologías utilizadas para implementar tanto la lógica de los procesadores como la de las memorias era bastante similar en materia de la frecuencia de trabajo. Sin embargo la frecuencia de trabajo de la lógica empezó a aumentar a un ritmo mayor que la de la memoria (en particular los circuitos de memoria utilizados para la memoria RAM principal).



Si bien en la última década la tecnología de memoria logró, por lo menos, no seguir perdiendo terreno, esto se debió básicamente a una ralentización del crecimiento de la frecuencia de trabajo de la lógica de los procesadores, motivada por ciertas barreras físicas difíciles de pasar con la tecnología de fabricación de circuitos integrados actualmente utilizada.

Este fenómeno de diferencia notoria en las frecuencias de trabajo fue la que llevó a la necesidad de mejorar y ampliar el uso de las memorias cache (tipo de memoria que desarrollaremos más adelante) y de hecho se pasó de una sola memoria de este tipo a tener dos (L1 y L2) y hasta tres niveles (L1, L2 y L3), como forma de mejorar el rendimiento de las computadoras sin elevar excesivamente el costo.

3 Principio de Localidad

La organización jerárquica de la memoria se basa en una característica que poseen la mayoría de los programas (al menos dentro de ciertos límites). Esta propiedad se denomina *principio de localidad*.

El principio de localidad establece que los programas acceden a una porción relativamente reducida del espacio de direcciones en un determinado lapso de tiempo.

El principio tiene dos variantes:

- Localidad temporal: si un ítem es referenciado en determinado momento, es común que vuelva a ser referenciado poco tiempo después.

- Localidad espacial: cuando un ítem es referenciado en determinado momento, es común que los ítems con direcciones "cercanas" también sea accedidos poco tiempo después.

Pensemos en las estructuras de control tipo bucle ó en el procesamiento de estructuras vectoriales o matriciales y veremos enseguida como se aplican estos principios. Por ejemplo en un "for" que recorra los elementos de un array tenemos por un lado que las instrucciones dentro del bucle del "for" van a ser referenciadas una y otra vez, cada cierto tiempo, mientras la cuenta no llegue el límite. Del mismo modo el dato que está a continuación del actual será utilizado luego (suponiendo que el array se recorre de a un elemento por vez). En este ejemplo se cumplen ambas variantes del principio de localidad.

¿Cómo aprovechamos el principio de localidad en la jerarquía de memoria?. Lo hacemos manteniendo los datos "recientemente accedidos" en las jerarquías altas, más cerca de quien lo consume (la CPU). También moviendo bloques de memoria contiguos hacia las jerarquías más altas. Con lo primero se apuesta a la localidad temporal, con lo segundo a la localidad espacial.

El principio de localidad es aplicado por distintos actores dentro de un sistema:

- registros <> memoria: lo aplica el compilador ó el programador de bajo nivel
- memoria cache <> memoria: lo aplica el hardware
- memoria <> disco: lo aplica el sistema operativo (memoria virtual) ó el programador (archivos)

4 Términos utilizados en Jerarquía de Memoria

Hay una serie de términos que se utilizan cuando se habla de jerarquía de memoria.

Hit

Se dice que ocurre un *hit* cuando un objeto de información se encuentra en el lugar de la jerarquía donde se lo está buscando.

Miss

Se dice que ocurre un *miss* cuando el elemento de información no es encontrado en el lugar de la jerarquía donde se lo está buscando. En este caso es necesario ir a buscar el objeto a un nivel de jerarquía inferior.

Hit Rate

Es la tasa de acierto de encontrar un elemento de información en el lugar de la jerarquía en que se lo busca.

Miss Rate

Es la tasa de fallos en encontrar un elemento de información en el lugar buscado (y coincide con1 – Hit Rate).

Hit Time

Tiempo de acceso promedio en el nivel de jerarquía considerado (donde se da el hit).

Miss Penalty

Tiempo de acceso promedio adicional requerido para acceder al elemento de información en el nivel de jerarquía inferior. Típicamente ocurre que Hit Time << Miss Penalty.

Tiempo promedio de acceso a memoria

Si consideramos el nivel de jerarquía "memoria cahé" podemos establecer que el tiempo promedio de acceso a memoria (es decir a elementos de información almacenados en la memoria principal de la computadora) cumple la relación:

Tiempo promedio de acceso a memoria = Hit Time + Miss Rate * Miss Penalty

5 Rendimiento de la Memoria

La velocidad de acceso a los elementos de información, incluyendo las instrucciones de los programas condiciona la capacidad de proceso de las computadoras. Es por esto que es el rendimiento del sistema de memoria tiene un impacto significativo sobre el rendimiento general.

Los sistemas de memoria tienen distintos parámetros que tienen relación con su capacidad de movilizar elementos de información:

Tiempo de Acceso: es el tiempo que transcurre entre que la dirección se presenta estable y los datos pueden ser manipulados en forma confiable. Puede haber diferencias entre el tiempo de acceso de lectura y el de escritura.

Tiempo de Ciclo: es el tiempo mínimo que debe pasar entre un acceso y el siguiente (las memorias requieren de un cierto tiempo de recuperación antes de poder iniciar un nuevo ciclo de acceso). Es la suma del tiempo de acceso más el de recuperación.

Tasa de Transferencia: es la velocidad de movimiento de datos de la memoria.

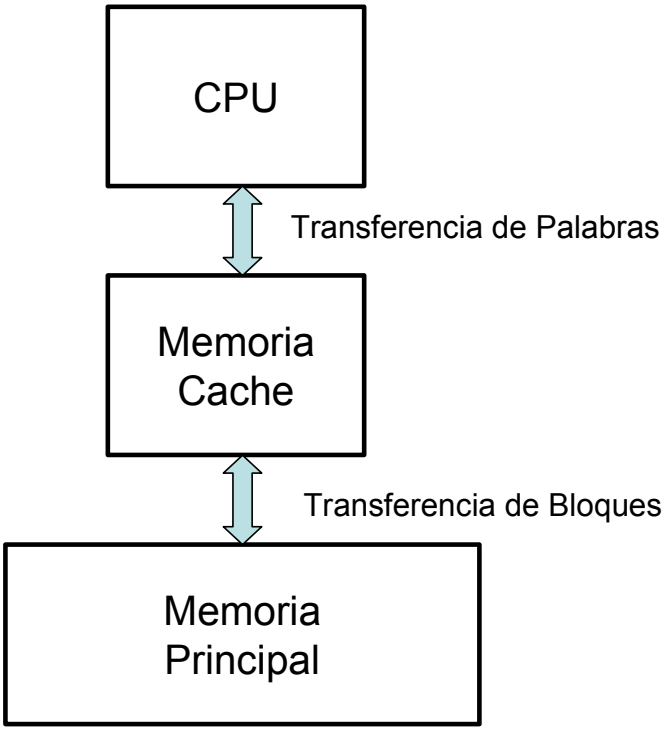
La tasa de transferencia está afectada por el tiempo de acceso y por el tiempo de ciclo, ya que ambos condicionan la velocidad con que los datos pueden ser obtenidos.

6 Memoria Caché

6.1 Introducción

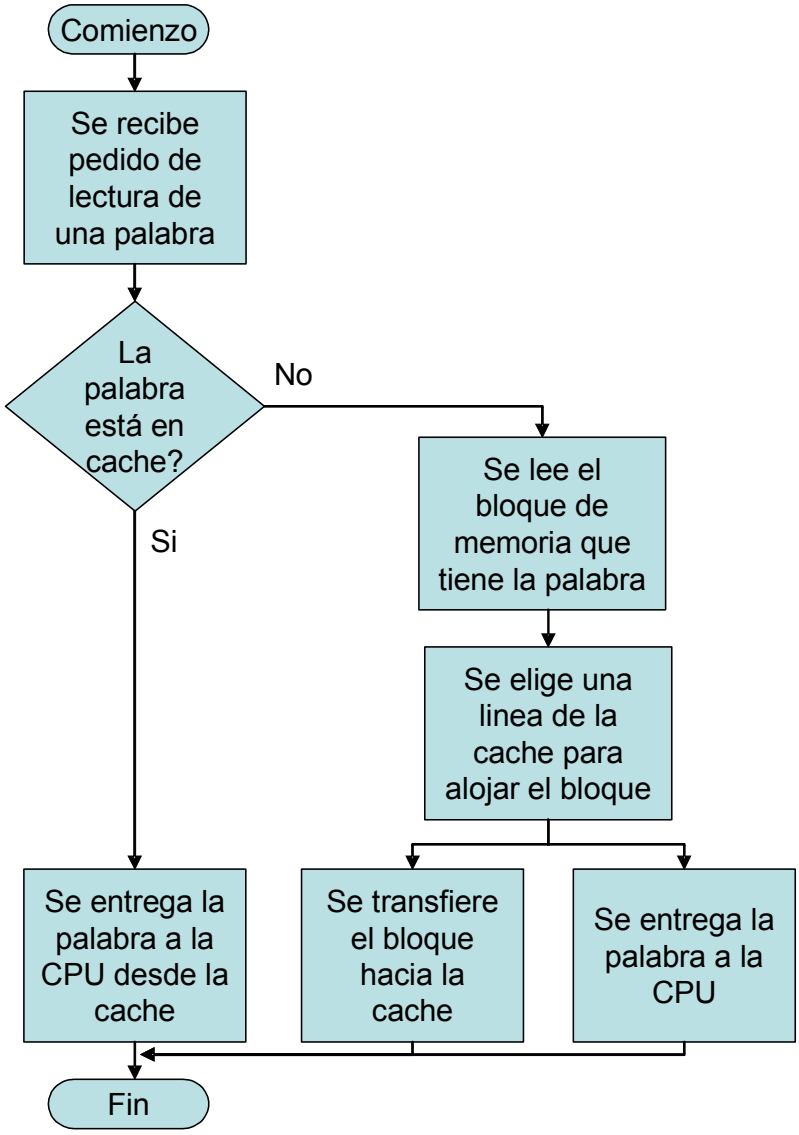
La memoria caché es una memoria de tamaño reducido, de alta velocidad, que se coloca entre la memoria principal y la CPU. Utilizando el principio de localidad mantiene copias de los bloques de memoria principal más accedidos, de manera que cuando la CPU requiere una palabra que está en uno de los bloques almacenados en la memoria caché, el

requerimiento es satisfecho desde la memoria caché, con un tiempo de acceso mucho menor que si debiera ser satisfecho desde la memoria principal.



Desde la memoria cache se accede a la memoria principal en bloques de ***k*** palabras (normalmente ***k*** > 1). Cuando un bloque es leído desde memoria se almacena en una ***línea*** de la cache.

La lógica de funcionamiento de la memoria cache es la que se esquematiza en el siguiente diagrama de flujo:



En definitiva si cuando la CPU requiere una palabra de memoria (para lectura) ocurre un “hit” en la memoria de mayor jerarquía (la cache) entonces el requerimiento se satisface desde dicha memoria, con un menor tiempo de acceso. Si por el contrario ocurre un “miss”, se dispara un mecanismo por el cual se trae desde la memoria el bloque que contiene la palabra buscada para simultáneamente escribirlo en la línea de la memoria cache que corresponda y entregar la palabra solicitada a la CPU.

En realidad luego veremos que es un poco más complejo el algoritmo, ya que al seleccionar una línea de la memoria cache para almacenar el bloque leído desde la memoria principal, puede significar la necesidad de escribir el contenido actual de la línea en el bloque de memoria principal correspondiente, dependiendo de la estrategia de “write” que se siga.

Notemos que cuando ocurre un “miss” (la palabra buscada no está en la memoria cahce) al realizar la transferencia desde memoria principal del bloque entero que contiene la palabra buscada hacia la memoria cache, estamos aprovechando el principio de localidad ya que lo mas probable es que el próximo requerimiento de acceso a la memoria sea una palabra próxima (seguramente la de la dirección siguiente), por lo que probablemente estará contenida en el bloque recién leído desde memoria. Es decir, luego de un “miss” lo

más probable es que ocurra un "hit" por la estrategia de mover bloques (y no palabras) desde la memoria principal a la memoria cache.

6.2 Diseño de la Cache

En el diseño de un sistema de memoria cache hay que tomar una serie de decisiones sobre distintos aspectos que tienen impacto sobre su rendimiento, entendido como el porcentaje de "hits" respecto al total de accesos.

Primero haremos un repaso de todos estos aspectos para luego detenernos en aquellos que merecen una explicación con mayor detalle.

Tamaño

Uno de los elementos a considerar es el tamaño de la memoria cache. Ya sabemos que la memoria utilizada para este subsistema es un elemento comparativamente muy caro, por lo que siempre deberemos lograr un compromiso entre la cantidad de memoria cache y la cantidad de memoria principal.

Si bien el punto de equilibrio depende fuertemente del tipo de programas que se ejecuten en el sistema, existirá una cierta cantidad de memoria a partir de la cual el incremento del rendimiento obtenido no compensa el costo adicional de agregar más memoria cache. Actualmente la memoria cache se coloca en el propio "chip" del procesador, por lo cual también existe una limitante adicional: la cantidad de transistores disponibles en el circuito integrado es finita y compite con todos los demás sub-sistemas que deben ser construidos.

El estado del arte actual de los procesadores comerciales disponibles indican que se utilizan 3 niveles de cache: el L1 de unos 10KB a 20KB, el L2 de 128KB a 512 KB y el L3 de 4MB a 12MB. Si consideramos que una computadora actual puede tener 4GB y más de memoria, vemos que la relación entre el tamaño de la memoria principal y la cache (L3) es de 1000 a 1.

Función de Correspondencia (Mapping)

Este elemento de diseño determina como se asocia una cierta posición de memoria con su posible ubicación en la memoria cache. La memoria cache está formada por un cierto conjunto de líneas. En cada línea se puede almacenar un bloque de memoria. Por tanto la función de correspondencia establece para cada bloque de memoria cuales son las líneas posibles de ser utilizadas en la cache para almacenarlo. La relación puede ser desde que cualquier línea de la cache puede recibir a un bloque dado de memoria (denominada correspondencia "completamente asociativa") hasta que solo una línea determinada puede recibir un bloque dado (denominada correspondencia "directa"), pasando por que cierto conjunto de *n* líneas puede alojar un bloque dado (es la correspondencia "asociativa por conjuntos de *n* vías").

Algoritmo de Sustitución

Al momento que ocurre un "miss" es necesario traer un nuevo bloque de la memoria principal a la cache. Para los casos en que hay más de un lugar posible de colocación del bloque (casos de correspondencia "totalmente asociativa" o "asociativa por conjuntos") es necesario tener un algoritmo para seleccionar cual de las líneas utilizar (y por tanto reemplazar su contenido en caso que no haya lugares libres, lo que va a ser la situación

habitual).

Política de Escritura

Hasta ahora hemos analizado como se comporta la memoria cache en la operación de lectura, pero también debemos analizar qué pasa cuando la CPU escribe en la memoria.

En las operaciones de escritura (*write*) existen dos grandes estrategias ***write through*** y ***write back***. La primera (*write through)* implica que, de existir un hit, la operación de escritura se hace en la memoria cache y también en la memoria principal. En la segunda en el caso del hit la escritura se realiza solamente en la memoria cache.

Naturalmente la estrategia *write back* permite un mejor desempeño del sistema en escritura, pero tiene la complejidad adicional de determinar si es necesario actualizar la memoria principal con el contenido de una línea de cahe que va a ser reemplazada como consecuencia de un miss, si es que el bloque tuvo algún cambio desde que se trajo a la cache. También tiene la tarea adicional de velar por la ***coherencia de la cache***, aspecto que veremos más adelante.

Tamaño del bloque

Un aspecto que debe analizarse es el tamaño del bloque (y por tanto de la línea del cache). Para un tamaño de memoria cache dado, ¿qué conviene más? ¿una línea más grande o más cantidad de líneas?.

En el primer caso por el principio de localidad la probabilidad de un *hit* luego de un *miss* aumenta, pero se puede sufrir ante cambios de contexto en ambientes de multiprogramación.

6.3 Función de Correspondencia Directa

La función de correspondencia directa (*Direct Mapping*) a cada bloque de memoria le asocia una única línea de cache. Esto significa que hay varios bloques de memoria que tienen la misma línea de cache asignada en la función de correspondencia.

Una forma de ver esta forma de correspondencia es que cada región de memoria principal de tamaño igual a la memoria cache puede estar contenida completamente en la memoria cache. Pero dos bloques ubicados en regiones distintas, separados un múltiplo del tamaño de la memoria cache competirán por la misma linea de la cache.

La principal ventaja de esta función de correspondencia es su simplicidad y por tanto su bajo costo de implementación. La desventaja es que un programa que se mueva por regiones de memoria muy distantes, posiblemente genere una gran cantidad de *misses* repercutiendo negativamente en el rendimiento de la memoria.

6.4 Función de Correspondencia Completamente Asociativa

En este caso la función de correspondencia (denominada *Fully Associative* en inglés) vincula un bloque de memoria con cualquier línea de la memoria cache. Es decir que un bloque dado puede estar en cualquier lugar de la cache.

6.5 Función de Correspondencia Asociativa por Conjunto de N Vías

En este caso la función de correspondencia (*N-way Set Associative* en inglés)

establece que cada bloque de memoria tiene asociado un conjunto de líneas de la memoria cache y puede estar almacenado en cualquiera de las líneas del conjunto. La cantidad de *vías* es la cantidad de líneas contenida en un conjunto. Ejemplo: una memoria cache asociativa por conjuntos de 4 vías tiene 4 líneas en cada conjunto. Un bloque dado podrá estar en cualquiera de las 4 líneas de su correspondiente conjunto.

6.6 Algoritmos de Sustitución

Cuando ocurre un *miss* y es necesario leer un nuevo bloque a la memoria cache, será necesario determinar cual línea ocupar, en caso que exista más de una posibilidad.

Para el caso de la función de correspondencia directa hay una sola línea posible por lo que este problema no existe. Pero para los casos de correspondencia totalmente asociativa o asociativa por conjuntos de n vías hay más de una posibilidad y se requiere definir la manera en que se determinará la selección.

Un aspecto a tener en cuenta cuando se piensan en estos algoritmos es que los mismos deben ser posibles de ser implementados en hardware, porque no pueden significar un impacto negativo en el rendimiento del sistema. Los algoritmos más utilizados son los siguientes:

Menos Recientemente Usado (LRU = Least Recently Used)

Este algoritmo selecciona para reemplazar, dentro de las líneas posibles, la que tenga el bloque que haya sido accedido hace más tiempo. Una implementación sencila de este algoritmo para una memoria cache asociativa por conjuntos de 2 vías es usar un bit que indica cual fue la última línea que se accedió. En cada acceso a un conjunto se actualizan los bits de ambas líneas del conjunto.

FIFO (First In First Out)

En este caso se selecciona la línea que contiene el bloque que haya sido traído primero desde la memoria principal (el más antiguo), sin importar si fue accedido ni que tantas veces lo fue. Una manera de implementar este algoritmo en hardware es mediante un buffer circular (con un puntero de circular por conjunto que señala la línea a reemplazar que se actualiza en cada reemplazo).

Menos Frecuentemente Utilizado (LFU = Least Frequently Used)

Este algoritmo utiliza la cantidad de veces que han sido accedidos los bloques de las líneas candidatas a ser reemplazadas. Para su implementación se pueden utilizar contadores en cada línea.

Random

El algoritmo random selecciona el bloque a reemplazar mediante una técnica aleatoria (normalmente pseudo-aleatoria por razones de implementación).