Algoritmos y estructuras de datos

Memoria, apuntadores y arreglos

Francisco Javier Zaragoza Martínez

Universidad Autónoma Metropolitana Unidad Azcapotzalco
Departamento de Sistemas

6 de mayo de 2015

Memoria y direcciones

La memoria de una computadora es un arreglo de celdas numeradas consecutivamente. Cada celda contiene un byte y está numerada por una dirección (de d bits). Las direcciones inician en la 0 y terminan en la $M - 1$ (donde $M = 2^d$ es el tamaño de la memoria).

Memoria y direcciones

La memoria de una computadora es un arreglo de celdas numeradas consecutivamente. Cada celda contiene un byte y está numerada por una dirección (de d bits). Las direcciones inician en la 0 y terminan en la $M - 1$ (donde $M = 2^d$ es el tamaño de la memoria).

Ejemplo (con $d = 4$)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
69	74	69	77	80	76	79	83	32	68	69	32	85	65	77	0

Interpretación de la memoria

Ejemplo (con $d = 4$)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
69	74	69	77	80	76	79	83	32	68	69	32	85	65	77	0

Como enteros de 4 bytes (`int`)

Los procesadores de Intel son *little endian*: El primer entero del ejemplo se calcularía como $69 + 74 \times 256 + 69 \times 256^2 + 77 \times 256^3 = 1296386629$:

bytes 0 a 3	bytes 4 a 7	bytes 8 a 11	bytes 12 a 15
1296386629	1397705808	541410336	5062997

Interpretación de la memoria

Ejemplo (con $d = 4$)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
69	74	69	77	80	76	79	83	32	68	69	32	85	65	77	0

Como caracteres (`char`)

El juego de caracteres estándar de C es ASCII:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E	J	E	M	P	L	O	S	' '	D	E	' '	U	A	M	'\0'

Interpretación de la memoria

Un ejemplo más complicado

Suponga que se hicieron las siguientes declaraciones de variables:

```
char a, b, c, d;
int  n, m;
char w, x, y, z;
```

entonces las variables `a`, `b`, `c` y `d` podrían ocupar las direcciones 0, 1, 2 y 3, las variables `n` y `m` las direcciones 4 a 7 y 8 a 11, y las variables `w`, `x`, `y` y `z` las direcciones 12, 13, 14 y 15.

Interpretación de la memoria

Un ejemplo más complicado

Suponga que se hicieron las siguientes declaraciones de variables:

```
char a, b, c, d;
int  n, m;
char w, x, y, z;
```

entonces las variables `a`, `b`, `c` y `d` podrían ocupar las direcciones 0, 1, 2 y 3, las variables `n` y `m` las direcciones 4 a 7 y 8 a 11, y las variables `w`, `x`, `y` y `z` las direcciones 12, 13, 14 y 15.

En este caso, los valores de las variables serían los siguientes:

```
char a = 'E', b = 'J', c = 'E', d = 'M';
int  n = 1397705808,   m = 541410336;
char w = 'U', x = 'A', y = 'M', z = '\0';
```

Operadores de referencia y desreferencia

Operador de referencia

El operador prefijo `&` obtiene la dirección de inicio de una variable. A esto también se le llama una referencia a la variable. Si declaramos `int n` entonces `&n` es una referencia a `n`.

Operadores de referencia y desreferencia

Operador de referencia

El operador prefijo `&` obtiene la dirección de inicio de una variable. A esto también se le llama una referencia a la variable. Si declaramos `int n` entonces `&n` es una referencia a `n`.

Operador de desreferencia

El operador prefijo `*` obtiene la variable a la que se hace referencia. A esto también se le llama una desreferencia. En otras palabras, `*&n` es `n`.

Direcciones y apuntadores

Direcciones

Las direcciones donde quedan almacenadas las variables en la memoria son determinadas por el compilador y el sistema operativo. Esto nos obliga a usar el operador `&` para obtener referencias a las variables.

Direcciones y apuntadores

Direcciones

Las direcciones donde quedan almacenadas las variables en la memoria son determinadas por el compilador y el sistema operativo. Esto nos obliga a usar el operador `&` para obtener referencias a las variables.

Variables de tipo apuntador

Un apuntador es una variable que puede almacenar una referencia a una variable de tipo específico. Los apuntadores se declaran así:

```
char *s, *t;
int  *p, *q;
```

Toda variable puede contener basura y los apuntadores no son excepción.

Ejemplo de uso de apuntadores

Ejemplo

Considere la declaración de variables y apuntadores:

```
int n=1, m=2;
int *p,  *q;
```

Ejemplo de uso de apuntadores

Ejemplo

Considere la declaración de variables y apuntadores:

```
int n=1, m=2;
int *p,  *q;
```

Si ahora hacemos

```
p = &n; /* direccion de n */
q = &m; /* direccion de m */
```

diremos que `p` apunta a `n` y que `q` apunta a `m` (`*p` es `n` y `*q` es `m`).

Ejemplo de uso de apuntadores

Ejemplo

Considere la declaración de variables y apuntadores:

```
int n=1, m=2;
int *p,  *q;
```

Si ahora hacemos

```
p = &n; /* direccion de n */
q = &m; /* direccion de m */
```

diremos que p apunta a n y que q apunta a m (*p es n y *q es m).

```
*p = 3; /* n vale 3 */
m = *p; /* m vale 3 */
q = p;  /* q apunta a n */
*q = 4; /* n vale 4 */
```

Funciones y referencias

Funciones en C

Todas las funciones en C reciben sus parámetros por valor. Esto significa que internamente se hace una copia y se trabaja con ella.

Funciones y referencias

Funciones en C

Todas las funciones en C reciben sus parámetros por valor. Esto significa que internamente se hace una copia y se trabaja con ella.

Funciones que modifican sus parámetros

Es evidente que algunas funciones deben poder modificar sus parámetros (y no copias de ellos). Por ejemplo, la función de biblioteca `scanf` debe poder leer datos de la entrada para colocarlos en variables del programa. Esto se hace así:

```
int n;
scanf("%d", &n);
```

Observe que `scanf` recibió una referencia a `n` (y no una copia de `n`).

Ejemplo (mal hecho)

Función de intercambio

```
void intercambia(int a, int b)
{
   int t;
   t = a;
   a = b;
   b = t;
}
```

Ejemplo (mal hecho)

Función de intercambio

```
void intercambia(int a, int b)
{
   int t;
   t = a;
   a = b;
   b = t;
}
```

¿Por qué no funciona?

Cuando llamamos a esa función de esta manera

```
int n=3, m=4;
intercambia(n, m);
```

lo que ocurre es que se hacen copias de `n` y `m` (ahora `a`=3 y `b`=4) y se intercambian los valores de `a` y `b` (ahora `a`=4 y `b`=3).

Ejemplo (bien hecho)

Función de intercambio

```
void intercambia(int *a, int *b)
{
   int t;
   t = *a;
   *a = *b;
   *b = t;
}
```

Ejemplo (bien hecho)

Función de intercambio

```
void intercambia(int *a, int *b)
{
   int t;
   t = *a;
   *a = *b;
   *b = t;
}
```

¿Por qué sí funciona?

Cuando llamamos a esa función de esta manera

```
int n=3, m=4;
intercambia(&n, &m);
```

lo que ocurre es que `a = &n` y `b = &m` y se intercambian los valores de `*a` y `*b` (es decir, de `n` y `m`).

Un arreglo en la memoria

Considere la declaración del arreglo `int a[4]` y el contenido de memoria:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
69	74	69	77	80	76	79	83	32	68	69	32	85	65	77	0

Como vimos antes, esto se interpreta como cuatro enteros de esta forma:

`a[0]` (0 a 3)	`a[1]` (4 a 7)	`a[2]` (8 a 11)	`a[3]` (12 a 15)
1296386629	1397705808	541410336	5062997

Memoria y arreglos

Un arreglo en la memoria

Considere la declaración del arreglo `int a[4]` y el contenido de memoria:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
69	74	69	77	80	76	79	83	32	68	69	32	85	65	77	0

Como vimos antes, esto se interpreta como cuatro enteros de esta forma:

`a[0]` (0 a 3)	`a[1]` (4 a 7)	`a[2]` (8 a 11)	`a[3]` (12 a 15)
1296386629	1397705808	541410336	5062997

Arreglos en la memoria

`T a[N]` declara un arreglo de `N` variables de tipo `T`. Si el tipo `T` requiere `B` bytes, entonces el arreglo `a` ocupa `N*B` bytes consecutivos.

Elementos de un arreglo

Dirección de los elementos

Si declaramos `int a[4]` entonces las direcciones de sus elementos son `&a[0]`, `&a[1]`, `&a[2]` y `&a[3]`.

El nombre del arreglo

El nombre del arreglo `a` es un sinónimo de la dirección de su primer elemento. De esta manera, hacer `p = &a[0]` es lo mismo que hacer `p = a`.

Moviéndose en un arreglo

Apuntadores e índices

Si declaramos `int *p` y `p = &a[0]`, entonces `p+1` apunta a `a[1]`, `p+2` apunta a `a[2]` y en general, si `i` es un `int`, entonces `p+i` apunta a `a[i]`. En otras palabras `p+i` vale `&a[i]` y `*(p+i)` es `a[i]`.

Moviéndose en un arreglo

Apuntadores e índices

Si declaramos `int *p` y `p = &a[0]`, entonces `p+1` apunta a `a[1]`, `p+2` apunta a `a[2]` y en general, si `i` es un `int`, entonces `p+i` apunta a `a[i]`. En otras palabras `p+i` vale `&a[i]` y `*(p+i)` es `a[i]`.

Incremento y decremento

Si `p` apunta a un elemento del arreglo, entonces `p++` hace que `p` apunte al siguiente elemento, mientras que `p--` hace que `p` apunte al anterior.

Moviéndose en un arreglo

Apuntadores e índices

Si declaramos `int *p` y `p = &a[0]`, entonces `p+1` apunta a `a[1]`, `p+2` apunta a `a[2]` y en general, si `i` es un `int`, entonces `p+i` apunta a `a[i]`. En otras palabras `p+i` vale `&a[i]` y `*(p+i)` es `a[i]`.

Incremento y decremento

Si `p` apunta a un elemento del arreglo, entonces `p++` hace que `p` apunte al siguiente elemento, mientras que `p--` hace que `p` apunte al anterior.

Comparación

Si `p` y `q` apuntan a elementos del mismo arreglo, entonces se pueden comparar. El resultado de la comparación es el mismo que el resultado de la comparación de los índices correspondientes.

Mínimo de un arreglo

Con arreglos e índices, regresando el valor

Esta función encuentra el menor valor almacenado en el arreglo `int a[]` de `n` elementos.

```
int minimo(int a[], int n)
{
  int i;
  int m = a[0];    /* el menor es el primero */

  for (i = 1; i < n; i++)
    if (a[i] < m)
      m = a[i];
  return m;
}
```

Mínimo de un arreglo

Con arreglos e índices, regresando el índice

Esta función encuentra el índice del menor valor almacenado en el arreglo `int a[]` de `n` elementos.

```
int minimo(int a[], int n)
{
  int i;
  int m = 0;          /* el menor es el primero */

  for (i = 1; i < n; i++)
    if (a[i] < a[m])
      m = i;
  return m;
}
```

Mínimo de un arreglo

Con arreglos e índices, regresando un apuntador

Esta función encuentra un apuntador al menor valor almacenado en el arreglo `int a[]` de `n` elementos.

```
int* minimo(int a[], int n)
{
  int i;
  int *m = &a[0]; /* el menor es el primero */

  for (i = 1; i < n; i++)
    if (a[i] < *m)
      m = &a[i];
  return m;
}
```

Mínimo de un arreglo

Con apuntadores, regresando un apuntador

Esta función encuentra un apuntador al menor valor almacenado en el arreglo que empieza en `int *a` y tiene `n` elementos.

```
int* minimo(int *a, int n)
{
  int i;
  int *m = a;       /* el menor es el primero */

  for (i = 1; i < n; i++)
    if (*(a+i) < *m)
      m = a+i;
  return m;
}
```

Mínimo de un arreglo

Sólo con apuntadores, regresando un apuntador

Esta función encuentra un apuntador al menor valor almacenado en el arreglo que empieza en `int *a` y tiene `n` elementos.

```
int* minimo(int *a, int n)
{
  int *i;
  int *m = a;       /* el menor es el primero */

  for (i = a+1; i < a+n; i++)
    if (*(a+i) < *m)
      m = a+i;
  return m;
}
```

Mínimo de un arreglo

Con dos apuntadores, regresando un apuntador

Esta función encuentra un apuntador al menor valor almacenado en el arreglo que empieza en `int *a` y tiene `n` elementos.

```
int* minimo(int *a, int n)
{
  int *b = a+n;    /* uno despues del final  */
  int *m = a;      /* el menor es el primero */

  for (a++; a < b; a++)
    if (*a < *m)
      m = a;
  return m;
}
```

Mínimo de un arreglo

Con dos apuntadores, regresando un apuntador

Esta función encuentra un apuntador al menor valor almacenado en el arreglo que empieza en `int *a` y tiene `n` elementos.

```
int* minimo(int *a, int n)
{
  int *b = a+n;    /* uno despues del final  */
  int *m = a;      /* el menor es el primero */

  while (++a < b)
    if (*a < *m)
      m = a;
  return m;
}
```

Ejercicios

Ordenando datos

1. Escribe una función `void dos(int *a, int *b)` que ordene `*a` y `*b` de modo que `*a <= *b`.
2. Escribe una función `void tres(int *a, int *b, int *c)` que ordene `*a`, `*b` y `*c` de modo que `*a <= *b <= *c`.

Mínimo y máximo

1. Escribe una función `int* minimo(int *a, int *b)` que regrese un apuntador al mínimo entero en el arreglo que empieza en `a` y acaba justo antes de `b`. Pista: modifica la última versión de `minimo`.
2. Reescribe todas las versiones de `minimo` para procesar el arreglo de atrás para adelante.
3. Reescribe todas las versiones de `minimo` pero para el máximo.