CC3301
Programación de Software de Sistemas
Profesor: Luis Mateu

- Big endian vs. little endian
- Alineamiento
- Una cola genérica
- Cast entre enteros y punteros
- Unboxing vs. boxing

Resumen clase pasada

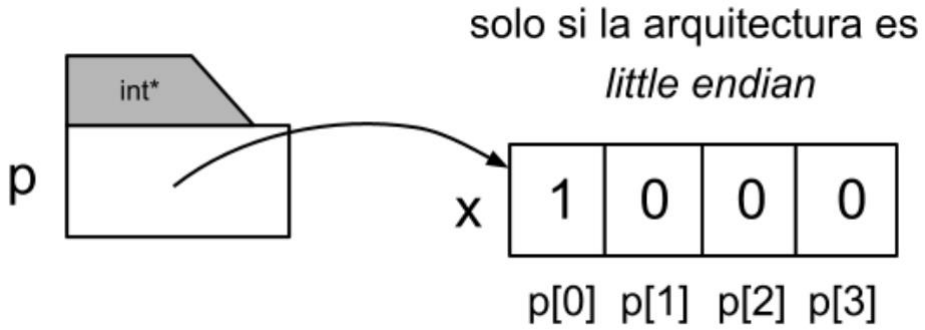
- Gracias a la *aritmética de punteros* y los *cast de punteros*, en C se puede almacenar datos de cualquier tipo en cualquier parte de la memoria del programa
- El *tipo dinámico* de una dirección de memoria es el tipo del último dato que se escribió en esa dirección
- Al leer ese dato por medio de un puntero es imprescindible que el *tipo estático* de su contenido sea idéntico al *tipo dinámico* de lo apuntado
- Pero abusar de los *cast de punteros* hace los programas muy frágiles y difíciles de depurar
- La mayoría del código no necesita los *cast de punteros*
- El *sistema de tipos de C* es un cinturón de seguridad que nos protege de errores accidentales en el manejo de punteros, pero no es una camisa de fuerza
- Los *cast de punteros* sirven en los pocos casos en donde el sistema de tipos de C es demasiado restrictivo, como en el caso de la función genérica para calcular integrales numéricamente, para las colecciones genéricas (mapas, colas, colas de prioridad, etc.), algoritmos genéricos (ordenamiento), etc.

Big endian vs. little endian

- ¿Qué valor retorna esta función?

```
int isLittleEndian() {
  int x= 1;
  char *p= (char*)&x;
  return p[0];
}
```

- Relación entre *x* y *p*:



- Depende del procesador: si es *little endian* significa que la dirección de una variable entera es la del byte menos significativo (*little end*) y por lo tanto retorna 1
- Si es *big endian* la dirección de una variable entera es la del byte más significativo (*big end*) y por lo tanto retorna 0
- X86, Arm y Risc-V son *little endian*
- Los procesador de IBM, Motorola y Sparc son *big endian*

Historia

- Viene de *los viajes de Gulliver* de Jonathan Swift (1726): los *big endians* son un grupo de gente de Lilliput que sostiene que los huevos duros se deben romper por el extremo más grande en vez del más pequeño, como lo ordenó el emperador de Lilliput (cita)
- Si un procesador es *little* o *big endian* es irrelevante desde el punto del área del chip, la frecuencia o el consumo: no es más caro ni más barato
- Pero sí es relevante si datos en formato binario se envían por la red o se almacenan en archivos: *si por ejemplo un procesador little endian envía por la red un dato binario y lo recibe un procesador big endian, el valor será incorrecto si no se hace la conversión*
- Sun Microsystems (ahora una división de Oracle) fue la empresa pionera de Internet y *ordenó* que todos los datos binarios transmitidos por Internet debían estar en formato *big endian*
- No hay un estándar para los archivos binarios

Alineamiento: ¿Qué muestra este programa?

```
typedef struct {
    char c;
    int n;
} U;

int main() {
    U *u= malloc(sizeof(U));
    printf("%p\n", u);
    printf("%ld\n", sizeof(U));
    printf("desplazamiento de c=%ld\n",
                                    (char*)&u->c-(char*)u);
    printf("desplazamiento de n=%ld\n",
                                    (char*)&u->n-(char*)u);
}
```

Salida:

0x56097c3262a**0**

8 // ¡Para garantizar el alineamiento de n!

desplazamiento de c=0

desplazamiento de n=**4**

Alineamiento

- **Definición**: Cuando las variables de tipos primitivos (int, short, double, etc.) y los punteros están *alineados* su dirección es múltiplo de su tamaño en bytes
- En algunas arquitecturas de procesadores acceder a datos no alineados gatilla un *bus error*
- En el resto de las arquitecturas, acceder a datos no alineados solo es *menos eficiente* porque requiere 2 accesos a la memoria, en vez de uno
- Los compiladores garantizan que variables locales y campos en estructuras locales que sean de tipo primitivo están alineados: el puntero a la pila siempre es una dirección múltiplo de 8 (al menos)
- Lo mismo ocurre con variables o arreglos dinámicos creados con malloc: las direcciones que retorna son siempre múltiplo de 8 (al menos)
- Si sizeof(U) fuese 5, no habría manera de alinear *n*
- Con el uso de cast de punteros y aritmética de punteros se puede hacer que un puntero direccione una variable no alineada

¿Qué muestra este programa?

```
typedef struct {
    int n;
    char c;
} T;

int main() {
    T *t= malloc(sizeof(T));
    printf("%p\n", t);
    printf("%ld\n", sizeof(T));
    printf("desplazamiento de c=%ld\n",
                                    (char*)&t->c-(char*)t);
    printf("desplazamiento de n=%ld\n",
                                    (char*)&t->n-(char*)t);
}
```

Salida:

```
0x56097c3262a0
8                       // ¡Para garantizar el alineamiento
                        // de n en arreglos de T!
desplazamiento de c=4
desplazamiento de n=0
```

Más sobre cast de punteros

- En clase auxiliar se implemento el tipo Queue:
  - Queue *makeQueue(); // Queue *q = makeQueue();
  - void put(Queue* q, char *str); // put(q, "perro");
  - char *get(Queue* q); // char *elem= get(q);
  - void freeQueue(Queue *q); // freeQueue(q);
- ¿Encolar enteros?

```
int a= 5, b= 7, c= -3;
Queue *q= makeQueue();
put(q, &a); put(q, &b); put(q, &c); // ✗ no compila
put(q, (char*)&a); put(q, (char*)&b); put(q, (char*)&c);
int *pa= (int*)get(q);                    // ✓
printf("%d\n", *pa);                      // ✓ ¡5!
```

- Funciona porque *Queue* nunca accede al contenido de los punteros depositados y el tipo estático de *pa* es consistente con su tipo dinámico

Una cola genérica

- Modificar Queue:
  - Queue *makeQueue(); // Queue *q = makeQueue();
  - void put(Queue* q, **void** *ptr); // put(q, "perro");
  - **void** *get(Queue* q); // char *elem= get(q);
  - void freeQueue(Queue *q); // freeQueue(q);
- ¿Encolar enteros?

```
int a= 5, b= 7, c= -3;
Queue *q= makeQueue();
put(q, &a); put(q, &b); put(q, &c); // ✔ sí compila
int *pa= get(q);                    // ✔ ¡sin cast!
printf("%d\n", *pa);                // ✔ ¡5!
```

- Se puede depositar cualquier cosa que sea un *puntero*
- Pero al extraer el contenido se debe usar un cast de puntero que coincida con el tipo dinámico de lo depositado
- Problema: ¿Qué pasa si se destruyen a, b o c?
- Los punteros almacenados en la cola se vuelven dangling references

¡Horror!

```
Queue *qabc(int a, int b, int c) { // ¡No haga esto!
    Queue *q= makeQueue(); // Es dinámica: se crea con malloc
    put(q, &a); put(q, &b); put(q, &c);
    return q;
}
int main() {
    Queue *q= qabc(5, 7, -3);
    int *pa= get(q);          // ✔ Hasta aquí vamos bien
    printf("%d\n", *pa);      // ✘ pa es dangling reference
}
```

```
int *makeInt(int n) { // Crea un entero en el heap: Boxing
    int *p= malloc(sizeof(int));
    *p= n;
    return p;
}
Queue *qabc(int a, int b, int c) { // Versión correcta
    Queue *q= makeQueue();
    put(q, makeInt(a)); put(q, makeInt(b)); put(q, makeInt(c));
    return q;
}
```

Cast entre enteros y punteros

```
Queue *qabc( ) {  // Unboxing
   int a= 5, b= 7, c= -3;
   Queue *q= makeQueue(); // Es dinámica: se crea con malloc
   put(q, (void*)a); put(q, (void*)b); put(q, (void*)c); // ?
   return q;
}
int main() {
   Queue *q= qabc();
   int a= (int)get(q);        // ?
   printf("%d\n", a);         // ✔ 5
   …
}
```

- ***Unboxing***: El cast (void*) disfraza un entero como si fuese un puntero
- warning: cast from pointer to integer of different size
- Hay warning solo si los punteros son de 64 bits
- No hay warning si los punteros son de 32 bits
- Pero ejecuta correctamente

Manera correcta de casts entre enteros y punteros para el unboxing

```
#include <stdint.h>
Queue *qabc( ) {
   intptr_t   a= 5, b= 7, c= -3;
   Queue   *q= makeQueue(); // Es dinámica: se crea con malloc
   put(q, (void*)a); put(q, (void*)b); put(q, (void*)c); // ✔
   return q;
}
int main() {
   Queue   *q= qabc();
   int  a= (intptr_t)get(q);      // ✔
   printf("%d\n", a);             // ✔ 5
}
```

- Se garantiza que el tipo *intptr_t* es del mismo tamaño que un puntero
- Está definido en *stdint.h*
- No hay warning ya sea que los punteros sean de 16, 32 o 64 bits
- No hay una manera estándar de disfrazar el tipo double

Comparación de unboxing vs boxing

- Unboxing es más cómodo y eficiente porque se evita el *malloc* y el *free*
- Pero es menos legible: no todos los programadores van a entender qué se está haciendo
- En los lenguajes con tipos dinámicos, como Python, las variables pueden almacenar valores de cualquier tipo
- Todas las variables son de tipo void* u Object (Java)
- La implementación estándar es que una variable es un puntero a una estructura en el heap de memoria, cuyo primer campo es una etiqueta con el tipo del valor almacenado y el siguiente campo es el valor
- En cada operación se chequea el tipo de los operandos
- *Boxing*: los enteros también se representan con punteros al heap, lo que es muy ineficiente
- *Unboxing*: los enteros (y reales opcionalmente) se almancenan directamente en el puntero de la variable
- Hay trucos para etiquetar los punteros que almacenan enteros y así distinguirlos del resto de los tipos de datos