

Manejando memoria dinámica

Heaps, mallocs y frees

Carlos Hernando

`chernando@acm.org`

ACM Facultad de Informática
Universidad Politécnica de Madrid

21 de noviembre de 2005
Curso de C 2005



Contenido

Memoria automática

- Introducción

- Limitaciones y problemas

Memoria manual

- Manejando memoria con malloc, free y realloc

- Problemas comunes



Reservando memoria en tiempo de compilación

Example

```
int test( int n ) {  
    int a, b;  
    a = 2;  
    b = 1;  
    return a * n + b;  
}  
...  
int r;  
r = test(10);
```

- ▶ Declaraciones: **tipo** e identificador.
- ▶ Su validez depende su **foco**:
 - ▶ Global
 - ▶ Locales
 - ▶ `static`
- ▶ La reserva y liberación de memoria es **automática**.

Memoria de un proceso

Memoria Proceso



Texto Código del programa.

Datos Datos del programa.

Pila Datos *temporales*.

Estructuras de datos que se crean y se destruyen,
por ejemplo las funciones.

Limitaciones y problemas

Tamaño estático Necesitamos saber el máximo de datos.

Especificación

... leer líneas del fichero ...

Paso de valores Generar datos dentro de una función.

Especificación

... randomblock:

Genera un array de enteros de tamaño indeterminado ...



Heap, diversión en memoria

Memoria Proceso



- ▶ Es la memoria del programa, viene y se va con él.
- ▶ En C, es el programador el encargado de su gestión.

malloc()

Queremos memoria

Definition

```
void *malloc(size_t size);
```

Example

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

► Conceptos: `void` y `cast`

- ▶ Reservamos un `size` de bytes.
- ▶ Consejo: `sizeof(tipo)`.
- ▶ Nos devuelve un puntero a cualquier cosa, **necesitamos un cast**.
- ▶ Si devuelve `NULL` malo.



free()

Devolvemos memoria

Definition

```
void free(void *ptr);
```

- ▶ Recibe el mismo puntero que `malloc()`.
- ▶ Si ya no necesitamos la memoria la liberamos.

realloc()

Queremos modificar el tamaño

Queremos modificar el **tamaño** de la memoria que hemos pedido.

Definition

```
void *realloc(void *ptr, size_t size);
```

- ▶ Es una mezcla entre `malloc()` y `free()`.
- ▶ Utilizamos el puntero de memoria y el tamaño deseado.
- ▶ Es muy recomendable comprobar que no devuelve *NULL*.



Punteros mal apuntados

1. Pedimos memoria.
2. Algo va mal:
 - ▶ Confundimos el tipo al pedir espacio.
 - ▶ Confundimos el cast.
 - ▶ malloc() falla pero no lo comprobamos.

Example

```
int * p;  
p = malloc( 8 * sizeof(char) );
```

3. Violación de segmento (SEGFAULT).



Goteos de memoria

1. Pedimos memoria.
2. Perdemos memoria:
 - ▶ Nos olvidamos de liberar la memoria.
 - ▶ Perdemos la referencia a ese bloque de memoria.

Example

```
int * p;  
p = (int *) malloc(5 * sizeof(int));  
p = (int *) malloc(4 * sizeof(int));
```

3. Somos malos programadores ;-)



Liberar demasiado

Cuestiones de **portabilidad**:

- ▶ Liberar dos veces:

Example

```
p = q;  
free(p);  
free(q);
```

- ▶ Liberar un NULL:

Example

```
p = NULL;  
free(p);
```

Liberar demasiado (y 2)

- ▶ Referenciar un espacio de memoria liberado:

Example

```
int *a, *b;  
int c;  
a = (int *) malloc(400);  
b = a;  
free(a);  
c = b[1];
```

El principal problema es que **suele funcionar**.



Resumen

Conocimientos adquiridos:

- ▶ Diferencias entre memoria automática y memoria manual.
- ▶ Funcionamiento de malloc(), free() y realloc().
- ▶ Problemas comunes con la gestión de memoria.

Conclusión:

- ▶ La memoria automática es más amable mientras que jugar con la memoria directamente es más flexible.
- ▶ Dejar al cargo del programador la gestión de la memoria exige mucha más responsabilidad.



Punteros genéricos y conversiones

void * y cast

- ▶ Punteros genéricos `void *`:
 - ▶ Un puntero es una dirección de memoria.
 - ▶ Un puntero genérico es un puntero sin significado.
- ▶ Conversión de tipos `cast`:
 - ▶ El fin es forzar la conversión de un tipo.

Sintaxis

```
destino = (tipo_destino) origen;
```

Example

```
int *n;  
n = (int *) p;
```

