2. Variables dinámicas

1. **Introducción**
2. **Gestión de memoria dinámica**
3. **Punteros y variables dinámicas en lenguaje algorítmico**
4. **Gestión de memoria dinámica y punteros en C**

Bibliografía

- **Biondi y Clavel.**
- **Kernighan y Ritchie.**

2.1. Introducción

- Variables estáticas:
  - Se crean con el algoritmo (programa/subprograma)
  - Existen (en la memoria del ordenador) mientras el algoritmo no termine
  - Se destruyen cuando el algoritmo termina
  - Se crean en tiempo de compilación

Ejemplos:

```
{Declaración de variables}
A, B: numérico;
C: carácter;
D: Vector[1..10] de cadena;
```

```
/* Variables estáticas */
int A; /* 4 bytes*/
float B; /* 8 bytes*/
char C; /* 1 byte*/
char C[10][20]; /* 200 bytes*/
```

- Las variables estáticas no permiten solucionar algunos problemas de forma eficiente:

Problema: ¿Cuáles serían las estructuras de datos necesarias para gestionar un índice de referencias de un libro?

Las características del texto son las siguientes:

– máximo de 200 páginas
– máximo de 300 referencias en el índice
– cada palabra aparece una media de 5 veces (y un máximo de 200)

Solución con las definiciones vistas hasta ahora:

```
{Declaración de Tipo}
INDICE = vector [1..300] de registro
                                referencia: cadena;
                                páginas: vector[1..200] de numérico;
                        fin registro
```

Espacio reservado (siempre):

300 x tamaño registro = 300 x (30 + 200)

Espacio (medio) necesario :

≤ 300 x tamaño de registro (medio) = 300 X (30 + 5)

- ¿Hay otras soluciones?
  – ¿Reservar memoria según la vamos necesitando?
  – Destruirla si no la necesitamos

uno	12	23							
dos	21	24	25						
tres	15	14							
:									
:									
quince	12	22	33	44	55	66	77	88	99
:									
:									
:									
cincuenta	1	299							

2.2. Gestión de memoria dinámica

Módulo de Gestión de Memoria Dinámica (MGMD)

- Existe un espacio de memoria reservado por programa
- Las variables estáticas ocupan sólo una parte de él
- MGMD: Gestiona espacio disponible de memoria por programa según las necesidades
- Permite crear/destruir objetos temporales
- Se crean/destruyen en tiempo de ejecución
- Estos objetos se denominan **Variables Dinámicas**
- Introducen nuevos problemas:
  - ¿cómo nombrar un número variable de objetos que pueden incluso no existir?
  - no se puede utilizar el esquema <nombre, tipo, valor>

2.3. Punteros en lenguaje algorítmico

2.3.1. Introducción

- Los punteros permiten hacer referencia a las variables dinámicas
  - son variables estáticas (tienen nombre, tipo y un valor)
  - su valor es el de la dirección de memoria de una variable dinámica (el nombre de la variable dinámica)
  - el operador → permite acceder al nombre de la variable dinámica
- Tenemos que distinguir:
  - P: variable tipo puntero: contiene una dirección
  - P →: contenido de la dirección de memoria a la que apunta P: valor de la variable dinámica

- Se utiliza **NIL** para designar un puntero nulo.
- El tipo de un puntero es fijo e indica al tipo de variable al que puede apuntar

- Ejemplo:

  P: **puntero a** carácter;

  Q: **puntero a** numérico;

RESUMEN:

- Un **objeto** de tipo **puntero** es un objeto **estático** que sirve para **referirnos a variables dinámicas**.
- Una **variable dinámica** es un objeto que **se crea** / **se destruye** durante la **ejecución** de un programa; tiene tipo y valor, **pero no nombre** (para lo que se usa el puntero junto con el operador →).

2.3.2. Creación y destrucción de variables dinámicas

Partimos de:

p: puntero de tipo T;
ok, lógico;

En general, el MGMD dispondrá de alguna versión de las siguientes primitivas:

- **intentar_obtener (p, ok)**
  p: puntero a T; {p. resultado}
  ok: lógico; {p. resultado}

  {El MGMD primero comprueba si hay espacio disponible; si la hay p apuntará a una variable de tipo T y ok será cierto; si no la hay, ok será falso}

A partir de la primitiva **intentar_obtener()**, tenemos que construir:

obtener (p, ok)
p: puntero a T; {p. resultado}
ok: lógico; {p. resultado}
Inicio
intentar_obtener(p, ok);
si no ok entonces
p := nil;
fin si
Fin

- En código algorítmico siempre se podrá obtener memoria dinámica, pero en general es necesario:

obtener(p);
si p <> nil entonces ...
sino Error ();
...

{p debe inicialmente apuntar a un objeto de tipo T}

- **intentar_liberar** (p, ok)
p: puntero a T; {p. dato-resultado}
ok: lógico; {p. resultado}
{Si p apuntaba a un objeto tipo T, se libera esa memoria y ok será cierto;
Si no apuntaba, ok será falso}

A partir de la primitiva **intentar_liberar()**, tenemos que construir:

liberar (p, ok)
p: puntero a T; {p. dato-resultado}
ok: lógico; {p. resultado}
Inicio
intentar_liberar(p, ok);
si ok entonces
p := nil; {continuar...}
sino Error();
fin si
Fin

Ejemplo:

¿cuál será el estado de la memoria de un programa tras las siguientes acciones?

{declaración de variables}

p, q, r: puntero a carácter;

c: carácter

0. inicio

1. obtener(p); obtener(q); obtener(r);
2. c := ‘x’;
3. p→:= ‘y’;
4. q→:= ‘z’;
5. r→:= c;
6. p→ := q→;
7. q→:= r→;
8. q := p;
   {perdemos una posición de memoria **frente a liberar(p)**}

9. fin

2. 4. Gestión de memoria dinámica y punteros en C

- Hasta el momento sólo se ha visto cómo el lenguaje C define y utiliza los punteros para acceder a las posiciones de memoria asignadas a un programa.
- No se ha tratado cómo “conseguir” nuevas posiciones de memoria (cómo funciona el Módulo de Gestión de la Asignación Dinámica de Memoria de C).

En la <stdlib.h> están definidas las siguientes funciones:

- void *calloc(size_t nobj, size_t size)
- void *malloc(size_t size)
- void *realloc(void *p, size_t size)
- void free (void * p)

- **void *calloc(size_t nobj, size_t size)**

  calloc obtiene (reserva) espacio en memoria para alojar un vector (una colección) de **nobj** objetos, cada uno de ellos de tamaño **size** bytes.

  Si no hay memoria disponible se devuelve NULL.

  El espacio reservado se inicializa a bytes de ceros.

  Obsérvese que calloc devuelve un **(void *)** y que para asignar la memoria que devuelve a un tipo puntero a **Tipo_t** hay que utilizar un operador de ahormado o cast: **(Tipo_T *)**

  Ejemplo:

  ```
  char * c;
  c = (char *) calloc (40, sizeof(char));
  ```

- **void *malloc(size_t size)**

  malloc funciona de forma similar a calloc salvo que:

  a) no inicializa el espacio, y
  b) es necesario saber el tamaño exacto de las posiciones de memoria solicitadas.

  Ejemplos de gestión de memoria dinámica en C:

```
/* declaración de variables
      s es un puntero a carácter,
      v es un vector de 10 punteros a carácter (posibles cadenas)*/
char *s, *v[10];

s = calloc(40, sizeof(char)); /*reserva 40 posiciones para caracteres */
scanf ("%40s", s); /* modifica s con una cadena*/
for (i=0; i<10; i++) {
      v[i] = (char *) malloc(40);
      gets(v[i]);
}.
```

- **void *realloc(void *p, size_t size)**

  realloc() cambia el tamaño del objeto al que apunta **p** y lo hace de tamaño **size**.

  El contenido de la memoria no cambiará en las posiciones ya ocupadas. Si el nuevo tamaño es mayor que el antiguo, no se inicializan a ningún valor las nuevas posiciones.

  En el caso en que no hubiese suficiente memoria para “realojar” al nuevo puntero, se devuelve NULL y p no varía.

  El puntero que se pasa como argumento ha de ser NULL o bien un puntero devuelto por malloc(), calloc() o realloc().

- **void free (void * p)**

  free() libera el espacio de memoria al que apunta p.

  Si p es NULL no hace nada.

  Además p tiene que haber sido “alojado” previamente mediante malloc(), calloc() o realloc().

```
/* EJEMPLO DE USO DE PUNTEROS EN C */
#include<stdio.h>
main(){
 /*Declaraci¢n de variables */
 char *s, *v[4];
 int i;
 /* Inicialmente s == NULL */
 printf ("\nInicialmente s es un puntero nulo:%ld y tamagno %d", s, sizeof(s));
 printf (" y el tamagno de *s %d", sizeof(*s));

 /* Reservamos memoria para s */
 s= (char *) calloc(40, sizeof(char)); /* equiv. (char *) malloc(40) */
 printf ("\nDespues, el tamagno de s es %d", sizeof(s));

 /* Obtenemos datos para s */
 printf ("\nDame un conjunto, menor de 40, caracteres: ");
 scanf ("%s", s);

 printf("\nDespues de leer, tamagno de s es %d,", sizeof(s));
 printf(" tamagno de *s es %d, y el contenido *s es %s",
   sizeof(*s) * strlen(s) , s);
}
```

```
for (i=0; i < 4; i++) {
   v[i] = (char *) malloc(40);
   if (v[i] != NULL) {
    printf ("\nDame una cadena: ");
    flushall();
    gets(v[i]);
    printf ("\nAcabo de leer: ");
    puts(v[i]);
   }
  }
```

{Otras funciones de E/S: getchar(), putchar(), getc(), putc()}

```
#define N 10
#include <stdio.h>
main(){
char c, *cambiante;
int i;

i=0;
cambiante = NULL;
printf("\nIntroduce una frase. Terminada en [ENTER]\n");
while ((c=getchar()) != '\n') {
  if (i % N == 0) {
   printf("\nLlego a %d posiciones y pido hasta %d", i, i+N);
   cambiante=(char *) realloc((char *)cambiante,(i+N)*sizeof(char));
   if (cambiante != NULL) {   /* Ya existe suficiente memoria para el
        siguiente carácter*/
     cambiante[i++] = c;
   }
  else
    exit(-1);
 } /* while*/
```

```
/* Antes de poner el terminador nulo hay que asegurarse de que haya suficiente memoria */
 if ((i % N == 0) && (i != 0)){
   printf("\nLlego a %d posiciones y pido hasta %d", i, i+N);
   cambiante=realloc((char *) cambiante, (i+N)*sizeof(char));
   if (cambiante == NULL) exit(-1);
 }
 cambiante[i]=0;
 printf ("\nHe leido %s", cambiante);
} /* main () */
```