Convenciones para proyectos

1 Introducción

Los proyectos se componen de diversos programas interrelacionados. Utilizar buenas prácticas de programación y pautar convenciones entre el equipo de trabajo son claves para facilitar el desarrollo, la verificación y el posterior mantenimiento del proyecto.
Además de las convenciones de código para el lenguaje C (estándar C99) sugeridas por la cátedra, se proponen a continuación algunas convenciones para proyectos.

2 Propuesta

2.1 Modularización

Dividir el código fuente en diferentes archivos (no escribir todo en único archivo `.c`).

La modularización mejora la organización del proyecto y facilita su comprensión. La división debe seguir una lógica, agrupando en un mismo archivo las partes del proyecto que estén mayormente relacionadas. La implementación de las estructuras de datos deben estar en archivos individuales.

2.2 Cabeceras

Utilizar archivos cabecera para llamar a funciones declaradas en otros archivos.

Los archivos cabecera (de extensión `.h`) contienen declaraciones y definiciones de macros y constantes para ser compartidas entre diferentes archivos fuentes. Por ejemplo, `stdio.h` contiene operaciones de entrada y salida de la librería estándar de C. También es posible definir archivos cabeceras propios, ver el Anexo 4 para un ejemplo.

2.3 makefile

Elaborar un makefile que genere automáticamente el ejecutable del proyecto.

El makefile contiene las directivas para compilar los diferentes archivos fuentes del proyecto. Para su eleboración, recomendamos leer el instructivo subido a Comunidades. Utilizar las siguientes banderas para compilar:

- -Wall y -Wextra: activan todos las advertencias
- -Werror: convierte las advertencias en errores
- -std=c99: usa el estándar C99
- -g: genera información para el debugging

2.4 Entrega

Guardar el proyecto en un directorio raíz titulado "ApellidoNombreTPx", donde x denota el número de TP. Incluir en él todos los archivos fuentes, cabeceras y el makefile necesarios para compilar el proyecto. El directorio raíz debe entregarse comprimido en formato `.tar` y los fuentes no deberán estar compilados (la compilación la realizará el docente a través del makefile correspondiente).

3 Debugging

No se aceptan programas con errores en el manejo de la memoria dinámica.

Recomendamos el uso de la herramienta Valgrind[1] para depurar los programas. Puede encontrar un instructivos con ejemplos en el siguiente enlace:
`http://www.it.uc3m.es/pbasanta/asng/course_notes/memory_profiler_es.html`

4 Anexo: Ejemplo archivo cabecera

A continuación se presenta un ejemplo de un archivo cabecera. Considerar el siguiente archivo `matriz.c` que implementa matrices mediante arreglos bidimensionales:

```
#include <assert.h>
#include <stdio.h>

struct Matriz_ {
  double** direccion;
  size_t numFilas;
  size_t numColumnas;
};

typedef struct Matriz_ Matriz;

Matriz* matriz_crear(size_t numFilas, size_t numColumnas) {
  Matriz* matriz = malloc(sizeof(Matriz));
  assert(matriz);

  matriz->direccion = malloc(numFilas * sizeof(double*));
  assert(matriz->direccion);

  for (size_t i = 0; i < numFilas; ++i) {
    matriz->direccion[i] = malloc(numColumnas * sizeof(double));
    assert(matriz->direccion[i]);
  }

  matriz->numFilas = numFilas;
  matriz->numColumnas = numColumnas;

  return matriz;
}

void matriz_destruir(Matriz* matriz) {
  assert(matriz);
  for (size_t i = 0; i < matriz->numFilas; ++i)
    free(matriz->direccion[i]);
```

[1] `https://valgrind.org/`

```
34   free(matriz->direccion);
35   free(matriz);
36 }
37
38 double matriz_leer(Matriz* matriz, size_t fil, size_t col) {
39   assert(matriz);
40   assert(fil < matriz->numFilas);
41   assert(col < matriz->numColumnas);
42   return matriz->direccion[fil][col];
43 }
44
45 void matriz_escribir(Matriz* matriz, size_t fil, size_t col, double val) {
46   assert(matriz);
47   assert(fil < matriz->numFilas);
48   assert(col < matriz->numColumnas);
49   matriz->direccion[fil][col] = val;
50 }
51
52 void matriz_imprimir(Matriz* matriz) {
53   assert(matriz);
54
55   for (size_t i = 0; i < matriz->numFilas; ++i) {
56     if (i)
57       printf("\n");
58     for (size_t j = 0; j < matriz->numColumnas; ++j)
59       printf("%lf ", matriz_leer(matriz, i, j));
60   }
61   printf("\n");
62 }
```

En virtud de llamar a estas funciones desde otros archivos, se define un archivo cabecera `matriz.h` con las declaraciones que se quieren exportar (y también definiciones de macros y constantes si las hubiera):

```
1 #include <stdlib.h>
2
3 typedef struct Matriz_ Matriz;
4
5 Matriz* matriz_crear(size_t numFilas, size_t numColumnas);
6
7 void matriz_destruir(Matriz* matriz);
8
9 double matriz_leer(Matriz* matriz, size_t fil, size_t col);
10
11 void matriz_escribir(Matriz* matriz, size_t fil, size_t col, double val);
12
13 void matriz_imprimir(Matriz* matriz);
```

Todas estas declaraciones pueden usarse externamente al incluir la directiva `#include "matriz.h"` en el archivo fuente. Notar el uso de `" "` para indicar que el archivo cabecera se encuentra en el directorio de trabajo. Por ejemplo se puede definir el siguiente archivo `main.c`.

```
1 #include "matriz.h"
2 #include <assert.h>
3 #include <stdio.h>
4
5 int main() {
6
7   Matriz* mat1 = matriz_crear(5, 10);
8
9   for (size_t i = 0; i < 5; ++i)
```

```
10      for (size_t j = 0; j < 10; ++j)
11        matriz_escribir(mat1, i, j, i);
12
13    printf("Matriz:\n");
14    matriz_imprimir(mat1);
15
16    return 0;
17  }
```

El preprocesador copiará el contenido de `"matriz.h"` y lo pegará en lugar de `#include "matriz.h"`. Notar que si el archivo cabecera fuese incluido más de una vez, el compilador procesaría su contenido más de una vez, causando posiblemente un error. Para evitar esto, es común envolver el archivo cabecera entre las directivas:

```
#ifndef __MATRIZ_H__
#define __MATRIZ_H__

/* resto del archivo cabecera */

#endif
```

De esta forma, si se incluye el archivo cabecera más de una vez, la condición pasaría a ser falsa y el preprocesamiento ignoraría por completo el contenido del archivo cabecera.

En general, es buena práctica mantener en los archivos cabecera únicamente declaraciones, y no definiciones, para evitar exponer la representación interna a otros programas. Como última aclaración, notar que el tipo `Matriz` quedó declarado en el archivo fuente y en la cabecera. Para evitar esto, lo correcto sería borrar el `typedef struct Matriz_ Matriz` de `matriz.c` (línea 10). Pero en ese caso, se le debe indicar al compilador que `Matriz` se encuentra ahora declarado en el archivo cabecera. Para ello, también se debe incluir en `matriz.c` la directiva `#include "matriz.h"`.