Explicación de la práctica 4

Structs y Modularización

Seminario de Lenguajes opción C

Facultad de Informática
Universidad Nacional de La Plata

2022

Indice

Structs y unions

- Son los equivalentes al "record" y "record case" de Pascal[1].
- Las structs permiten agrupar variables de distintos tipos, son fundamentales para definir nuevos tipos de datos.
- Los miembros de las unions en cambio, ocupan el mismo espacio de memoria, por lo que solo debemos acceder a uno de los miembros (el resto tendrá basura).
- Las unions no son usadas comunmente en C, nos enfocaremos en las structs.

[1] https://en.wikipedia.org/wiki/Comparison_of_Pascal_and_C

Structs

- Hay distintas formas de declararlas:

```
struct nodo_lista{
    int dato;
    struct nodo_lista *siguiente;
};
typedef struct{
    char *nombre;
    char *apellido;
} persona_t;
```

- Dependiendo de como esté declarada deberemos usar o no la palabra struct al definir variables de ese tipo:

```
struct nodo_lista mi_nodo;
persona_t mis_datos;
```

Structs

Acceso a los miembros de structs

- Si una variable es de tipo struct, usaremos . para acceder a sus miembros:

```
mi_nodo.dato = 5;
mis_datos.nombre = malloc(strlen("pepe") + 1);
strcpy(mis_datos.nombre, "pepe");
```

- En cambio si una variable es de tipo puntero a struct, debemos usar -> para acceder a sus miembros:

```
struct nodo_lista *lista;
lista = malloc(sizeof(struct nodo_lista));
lista->dato = 5;
lista->siguiente = NULL;
```

Structs

Tamaño de los structs

- Un struct es al menos tan grande como sus miembros, es decir:
  `sizeof(struct nombre) >= sizeof(char *) * 2`
- Después de cada miembro de un struct pueden haber bytes de relleno para que los miembros queden alineados en memoria.
- Por esto hay que tener cuidado al cargar los datos de un struct desde un archivo. En general: cargar y guardar los structs de a un miembro.

Modularización

Objetivos de la modularización

- Legibilidad.
- División de trabajo.
- Creación de bibliotecas.
- Hacer manejable un proyecto grande.

Cómo hacerlo correctamente

- Agrupar el código por funcionalidad.
- Agrupar por estructuras de datos.
- Dividir módulos grandes en otros más manejables.
- Interfaz separada de implementación.
  - Interfaz $\rightarrow$ declaraciones (no generan código ni reservan espacio de memoria).
    - Prototipos de funciones.
    - Macros.
    - typedefs.
    - Declaración de variables "extern".
  - Implementación $\rightarrow$ definiciones (generan código y/o reservan espacio en memoria).
    - Definiciones de variables (static/extern/automáticas).
    - Código de funciones.

Visibilidad

Se puede cambiar la visibilidad de un objeto externo (global):

- `static` → Visible solamente en el archivo o bloque actual.
- `extern` → Visible a quienes incluyan esta declaración.

Las funciones y variables globales siempre son `extern` a menos que se declaren `static`.

Visibilidad: Ejemplo

visibilidad.c

```
static int func1() { ... } // Local a este archivo
int func2() { ... } // Extern
int x; // Extern
static int y; // Local a este archivo
```

visibilidad.h

```
static int func1(); // No tiene sentido...
int func2(); // Permite usarla en otros .c
extern int x; // Lo mismo
extern int y; // Es un error...
```

Static en una variable interna

Las variables automáticas se alocan al entrar en un bloque y se liberan al salir.
Se puede tener una variable local no automática con `static`, su tiempo de vida será desde que se ejecuta hasta que termina el programa.

funcs.c

```
void funcion(){
    static int x = 0; // Se va incrementando
    int y = 0; // Sólo llega hasta 1 y se pierde
    x++; y++;
    printf("x = %d, y = %d\n", x, y);
}
```

Doble inclusión

El preprocesador copia código por cada include que hagamos. Ejemplo trivial:

vector.h

```
typedef struct vector {
        void *datos;
        unsigned
            tam_logico;
        unsigned
            tam_alocado;
} vector_t;
/* ... */
```

programa.c

```
#include "vector.h"
#include "vector.h"
/* ... */
```

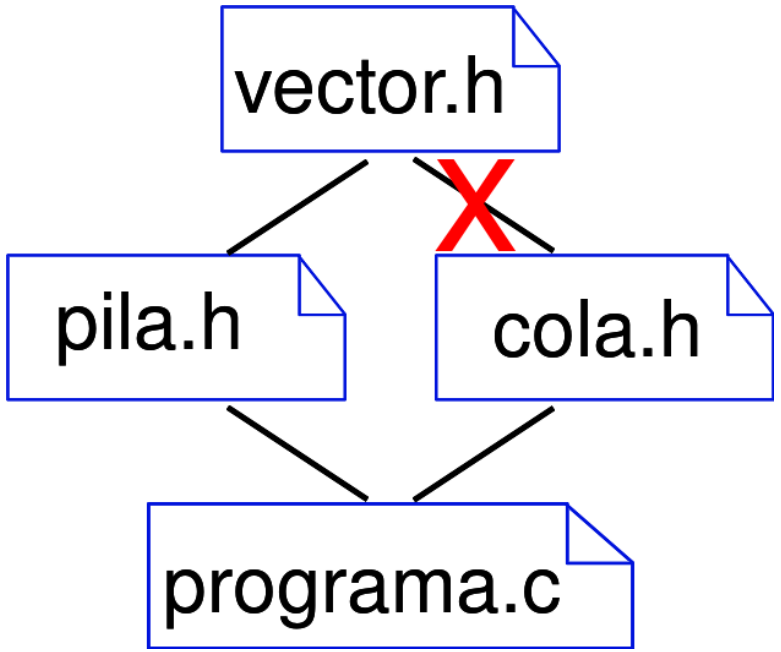
preprocesado.E

```
typedef struct vector {
```

Ejemplo más complicado

Suponer el caso de un programa que usa pilas y colas. Ambas implementadas con un vector.

¿Cómo evitar la doble inclusión sin que el usuario de los .h se tenga que preocupar?



Evitar la doble inclusión

Existe un mecanismo para evitar la doble inclusión sin que el usuario de los .h se entere:

```
#ifndef PILA_H
#define PILA_H
// la forma PILA_H es una convención, pero
// podría ser cualquier nombre...
// Declaraciones normales del .h ...
#endif
```

stdint.h parte de la librería estándar

```
#ifndef _STDINT_H
#define _STDINT_H
// el contenido de stdint.h ...
#endif /* stdint.h */
```