Modularización

1. Motivación de la separación de archivos

Cuando la dimensión o complejidad del programa a construir crece es conveniente dividir el código en varios archivos, en cada uno de los cuales se incluyen entidades (tipos, constantes, funciones) altamente relacionadas entre sí. A estos agrupamientos se les suele denominar *módulos*. Con esta división es posible que el trabajo sea realizado de manera independiente en cada uno de los módulos, posiblemente por distintos desarrolladores. Además, algunos de esos módulos pueden reusarse para otros proyectos, o sea, servir como bibliotecas.

La división en módulos requiere que se pueda cumplir con la compilación separada, esto es, que cada uno de los módulos pueda ser compilado sin disponer del código de los otros. Para lograr esto se presenta un problema cuando un módulo depende de entidades que serán implementadas en otros, ya que deben incluirse para poder compilar pero no se dispone de ellas. Este se soluciona separando cada módulo en dos archivos, en uno de los cuales se declaran las entidades y en el otro se las implementa. En el lenguaje `C` a los primeros se los suele identificar con la extensión *.h* (o, en `C++`, también *.hpp*) (la letra proviene de *headers*, encabezamientos).

Supongamos que para la implementación de `TPila` se usa el tipo `TLista` que en consiste de listas hasta `MAX` elementos, donde `MAX` es una constante definida en `utils.h`. El tipo `TLista` es definido en `lista.h` junto con la declaración de las operaciones que se pueden hacer con el tipo:

```
// en lista.h
typedef struct _rep_lista * TLista;
```

Con esto se establece que los elementos de tipo `TLista` son punteros a elementos de tipo `struct _rep_Lista`. Este último tipo aún no ha sido definido (se debe definir en `lista.cpp`).

Para poder desarrollar e compilar `TPila` se incluyen en `pila.cpp` los archivos de encabezamiento:

```
// en pila.cpp
#include "include/lista.h"
```

Aquí se está suponiendo que `lista.h` está en el subdirectorio `include` y `pila.cpp` está en algún otro subidirectorio, por ejemplo `src`. La directiva `#include` hace que en la fase de preprocesamiento de la compilación la línea sea sustituida por el contenido de `lista.h`, lo que permite que las entidades ahí declaradas sean usadas en el resto del archivo `pila.cpp`. De este modo se puede compilar y obtener `pila.o` aunque no se disponga de `lista.cpp`:

```
$ gcc -Wall -Werror -Iinclude -c src/pila.cpp -o obj/pila.o
```

Con las opciones `-Wall -Werror` se logra que todas las advertencias (warnings) sean tratadas como errores. La opción `-c` hace que solo compile, esto es, que traduzca el código `C` a código objeto, sin generar un ejecutable. Mediante la opción `-o` se establece la ruta y el nombre del código objeto.
Se debe notar que aunque no se conoce la implementación de `struct _rep_lista` se sabe que las variables de `TLista` son de tipo puntero, por lo que se puede determinar el espacio de memoria que necesitan.

2. Implementación

2.1. Lista

Lo que se declara en `lista.h` se debe implementar en `lista.cpp`, por lo que en este archivo se debe incluir el anterior:

```
#include "../include/lista.h"
```

El archivo se compila mediante:

```
$ gcc -Wall -Werror -Iinclude -c src/lista.cpp -o obj/lista.o
```

Entre lo que se debe implementar está la representación de `TLista` que es `struct _rep_lista`. Por ejemplo

```
/*
 Se define la representación declarada en lista.h
  Es un arrglo con tope,
   En C los índices de los arreglos empiezan en 0.
    Aquí no se va a usar la celda 0 por lo cual el tamaño del arreglo
     debe ser MAX + 1.
      Los elementos de la lista están entre 1 y 'longitud'.
 */
 struct _rep_lista {
       nat longitud;
         info_t infos[MAX + 1];
         };
```

(el tipo `nat` se habría definido previamente, en `utils.h`).

El tipo `struct _rep_lista` queda definido pero solo dentro de `lista.cpp` (un intento de usarlo en otro módulo generaría el error de tipo incompleto).

Entre las operaciones del módulo *lista* algunas devuelven elementos de tipo `TLista`. Como estos son punteros la memoria a la que apuntan se obtiene de manera dinámica. Esto se hace con el operador `new`.

```
/*
  Se obtiene un bloque de memoria para alojar el struct _rep_lista.
    res es una referencia al bloque de memoria donde se aloja el registro.
      (*res) es la desreferencia de res: es el contenido referenciado por ↩
          res.
  */
  TLista crearLista() {
        TLista res = new _rep_lista;
          (*res).longitud = 0;
            // res->longitud = 0; // equivalente a lo anterior
              return res;
          }
```

Cuando ya no se necesita usar la variable se debe liberar (devolver) la memoria asignada, lo que se hace mediante el operador `delete`:

```
void liberarLista(TLista lista) {
      delete lista;
      }
```

2.2. Pila

La implementación de `TPila` se puede hacer mediante `TLista`

```
// Se define la representación de TPila.
```

```
// Se usan TLista y sus operaciones.
struct _rep_pila {
      TLista lst;
      };
```

Hay que notar que, aunque tiene un solo campo, el mecanismo de encapsulamiento de `C` necesita que esté contenido en un *struct*.

Para devolver elementos de tipo `TPila` también hay que obtener memoria de manera dinámica:

```
TPila crearPila() {
      // Se obtiene memoria para la representación de la pila.
        TPila resultado = new _rep_pila; // C++

          // (*resultado) es un registro de tipo repPila.
        // Su único campo es lst, que debe ser inicializado.
          (*resultado).lst = crearLista();
            // De manera alternativa se puede usar el operador '->':
              // resultado->lst = crearLista();
            return resultado;
            }
```

Se debe notar que se obtiene memoria en dos instrucciones. Una de manera directa para `_rep_pila` y otra de manera indirecta al inicializar la `TLista` contenida en el *struct*.
Esto se refleja al liberar la memoria:

```
void liberarPila(TPila pila) {
      liberarLista((*pila).lst);
        delete pila;
        }
```

El uso del operador `delete` es similar a como se hizo en `pila.src`. Pero además hay que liberar la memoria asignada de manera dinámica a sus componentes, en este caso `lst`. Esto no pasó en `liberarLista` porque los componentes de `_rep_lista` usan memoria estática.

Las operaciones de `TPila` se implmentan usando las operaciones de `TLista`

```
info_t cima(TPila p) {
      assert(!esVaciaPila(p));
        return infoLista(longitud((*p).lst), (*p).lst);
        }
```

En la implementación de `cima` se usan las operaciones `longitud` e `infoLista` de `TLista`.
No se puede implementar usando la representación de `TLista`

```
   return (*p).lst).infos[((*p).lst).longitud];
```

porque provocaría un error de compilación ya que el campo `infos` no es conocido en este ámbito.

3. Directivas ifndef, define, endif

En principio podría ocurrir que un archivo se incluya varias veces. Por ejemplo `pila.cpp` puede incluir también `utils.h`, que también debió ser incluido en `lista.h` para acceder a la constante `MAX`. :

```
// en lista.h
#include "utils.h"
```

```
// en pila.cpp
#include "include/utils.h"
#include "include/lista.h"
```

Como consecuencia las entidades declaradas en `utils.h` se incluirían dos veces en el código final: la primera de manera directa y la segunda a través de la inclusión de `lista.h`.

Para evitar esto se usan las directivas `#ifndef`, `#define`, `#endif` en los archivos con extensión `.h`.

El archivo `utils.h` se vería así:

```
#ifndef _UTILS_H
#define _UTILS_H

#define MAX 10 // cantidad máxima de elementos de las colecciones

// otras declaraciones

#endif
```

La primera vez que se intenta incluir `utils.h` aún no se ha definido `_UTILS_` por lo que se procesa el resto del archivo, lo que incluye definir `_UTILS_`. En los siguientes intentos de incluir `utils.h` al procesar la directiva `#ifndef _UTILS_` se comprueba que ya se encuentra definida `_UTILS_` por lo que se omite todo lo que hay hasta `#endif`.

Estas directivas NO deben usarse en los archivos de implementación.