TADs en C

Matías Bordese
Algoritmos y Estructuras de Datos II - Laboratorio 2013

1. Objetivos

- Definición de TADs en C
- Ocultación de la implementación del TAD
- Manejo básico de memoria dinámica

2. Usando struct

Dado el tipo `tperson` similar al visto en el teórico, construido a partir de `tuple`, se quiere ver cómo se implementaría en el lenguaje de programación C.

```
type tperson =
    tuple
        id: nat
        age: nat
        weight: real
    end

var p: tperson
```

Lo que el teórico se define como un `tuple`, en C se mapea a una estructura en C. Para definir una estructura en C se utiliza el constructor `struct`, que espera el tipo y nombre de los campos que componen la estructura. Para el ejemplo de arriba nos quedaría algo como lo siguiente:

```
struct _person_t {
    unsigned int id;
    unsigned int age;
    float weight;
};

struct _person_t p;
```

Para declarar una variable p de tipo persona, hay que usar la sentencia arriba especificada, *NOTAR* que el nombre completo y correcto de este nuevo tipo de datos es `struct _person_t`.

2.1. Y esto cómo se usa?

Por ejemplo, definamos una función que crea una nueva instancia de persona, y otra que imprime los datos de una persona por pantalla:

```
struct _person_t person_create(unsigned int id, unsigned int age, float weight) {
    struct _person_t person;

    person.id = id;
    person.age = age;
    person.weight = weight;

    return(person);
}

void print_person(struct _person_t person) {
    printf("Person %d\n", person.id)
    printf("\tage: %d, weight: %f\n", person.age, person.weight);
}

...

struct _person_t p;
p = person_create(1, 23, 80.5);
print_person(p);
```

2.2. Organizando el código

Llevando estos conceptos a código "compilable", podríamos tener los siguientes archivos:

[person.h]

```
struct _person_t {
    unsigned int id;
    unsigned int age;
    float weight;
};

struct _person_t person_create(unsigned int id, unsigned int age, float weight);
void print_person(struct _person_t person);
```

[person.c]

```c
#include <stdio.h>
#include "person.h"

struct _person_t person_create(unsigned int id, unsigned int age, float weight) {
    struct _person_t person;

    person.id = id;
    person.age = age;
    person.weight = weight;

    return(person);
}

void print_person(struct _person_t person) {
    printf("Person %d\n", person.id);
    printf("\tage: %d, weight: %f\n", person.age, person.weight);
}
```

[main.c]

```c
#include "person.h"

int main(void) {
    struct _person_t p;
    p = person_create(1, 23, 80.5);
    print_person(p);

    return(0);
}
```

La idea detrás de un TAD es definir un tipo y una interfaz reusable, general y flexible, de tal manera de poder distribuir nuestro código compilado (un archivo .o, por ejemplo) y la descripción de la interfaz (el archivo .h) para quien quiera reutilizar nuestro TAD (incluso nosotros mismos en otro proyecto). Para llegar a este punto, aún hay algunas cuestiones por mejorar.

3. Usando typedef

Algo molesto del código en su estado actual es el hecho de tener que hablar del tipo `struct _person_t`. Es un poco largo, y propenso a producir equivocaciones.

Existe una manera de definir sinónimos de tipos, usando `typedef`:

```
typedef [nombre original] [nuevo nombre];
```

En nuestro caso, podemos hacer:

```
typedef struct _person_t person_t;
```

3.1. Actualizando el código

[person.h]

```
struct _person_t {
    unsigned int id;
    unsigned int age;
    float weight;
};

typedef struct _person_t person_t;

person_t person_create(unsigned int id, unsigned int age, float weight);
void print_person(person_t person);
```

[person.c]

```
#include <stdio.h>
#include "person.h"

person_t person_create(unsigned int id, unsigned int age, float weight) {
    person_t person;

    person.id = id;
    person.age = age;
    person.weight = weight;

    return(person);
}

void print_person(person_t person) {
    printf("Person %d\n", person.id);
    printf("\tage: %d, weight: %f\n", person.age, person.weight);
}
```

[main.c]

```c
#include "person.h"

int main(void) {
    person_t p;
    p = person_create(1, 23, 80.5);
    print_person(p);

    return(0);
}
```

Todavía tenemos un inconveniente por resolver. No queremos que nadie, a excepción de la implementación de `person_t` mismo, conozca la definición de la estructura y pueda usar los campos como le parezca, si no que utilicen nuestro TAD a partir de una interfaz que definamos nosotros (esto implica que se debe proveer una interfaz completa y útil, con los accesores y modificafores necesarios).

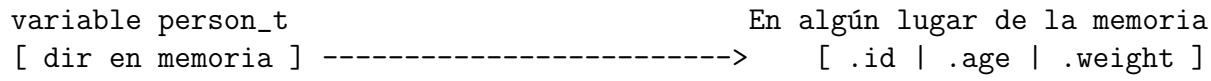
Como vemos arriba, la definición de la estructura está en el archivo .h, que es el que uno distribuiría con su librería. El próximo paso será ocultar esta definición.

4. Cómo ocultar la estructura

No es tan simple como mover la definición del struct al .c, ya que en el .h hacemos referencia al tipo que estamos definiendo. Pero podemos reformular el sinónimo para que en lugar de sólo cambiar el nombre, convertir el tipo `person_t` en un puntero a un `struct _person_t`:

```c
typedef struct _person_t *person_t;
```

Con esto estamos definiendo un tipo `person_t`, que es un puntero a una estructura `struct _person_t`. Es decir el tipo `person_t` ahora en lugar de almacenar todos los campos, sólo almacena una dirección de memoria, en la cual se encuentra la estructura y los correspondientes campos:



Qué ganamos con esto? Ahora `person_t` es en realidad un tipo de puntero, y el compilador ya no necesita conocer la estructura por dentro, ya que una variable de tipo `person_t` es una dirección (y este tipo ya lo conoce, y sabe cuánta memoria requiere).

Pero ahora, para construir una instancia de persona debemos pedir espacio en memoria para una estructura `struct _person_t` y guardar en una variable de tipo `person_t` (que guarda direcciones de memoria a estructuras `struct _person_t`) la dirección que se nos asignó para almacenar nuestra instancia.

4.1. Constructor

```
person_t person_create(unsigned int id, unsigned int age, float weight) {
    person_t person = NULL;

    person = calloc(1, sizeof(struct _person_t));
    assert(person != NULL);

    person->id = id;
    person->age = age;
    person->weight = weight;

    return(person);
}
```

Lo de arriba se podría usar de la siguiente manera:

```
person_t p = NULL;
p = person_create(1, 23, 80.5);
```

La función `calloc` toma dos argumentos: un entero, indicando la cantidad de items que planeo guardar en memoria; y otro entero, el tamaño del tipo a guardar. Nos devuelve una dirección de memoria, en la cual se nos asignó el espacio solicitado. En caso de no haber memoria disponible, devuelve `NULL`, valor que corresponde a la dirección nula.

En nuestro caso pedimos memoria para 1 item del tamaño de la estructura `struct _person_t` (la función `sizeof` nos calcula el tamaño automáticamente si le pasamos el tipo de la estructura). La dirección a ese espacio en memoria queda almacenado en la variable `person`. Notar como ahora inicializamos las variables de tipo `person_t` a `NULL`.

A partir de ese momento, podemos usar la variable `person` *casi* como una estructura normal, salvo que tenemos que indicarle al compilador que ahora para acceder a la estructura tiene que seguir un puntero (o dirección). Es por eso que para referenciar los campos dentro de la estructura ahora usamos `->` en vez de la notación con punto.

La notación `person->id` es equivalente a `(*person).id`, es decir, seguir la dirección a la que apunta `person` (`*person`), y una vez en esa dirección, acceder al campo `id`. Esto es válido porque como hemos definido, por su tipo, `person` es un puntero a una estructura `struct _person_t`; entonces, cuando llego a lo apuntado por `person`, lo que hay allí es un estructura `struct _person_t`, y puedo accederla como uno esperaría (como vimos al principio).

Algo más a tener en cuenta es que estamos pidiendo memoria en forma dinámica, es decir, durante la ejecución de nuestro programa (o sea, en cualquier momento que nuestro programa decida crear una instancia de persona tiene que pedirle un espacio en la memoria al sistema operativo). Nuestro programa debería ser un "buen ciudadano" y devolver esa memoria al sistema operativo cuando ya no la use.

4.2. Destructor

```
person_t person_destroy(person_t person) {
    free(person);
    person = NULL;
    return(person);
}
```

Lo de arriba se usa de la siguiente manera:

```
p = person_destroy(p);
```

Notar como al llamar al destructor, éste devuelve el valor `NULL`, valor que asignamos a la misma variable que acabamos de destruir, de tal forma de asegurarnos que el valor final de esta variable (que tiene que ser una dirección), sea el de la dirección nula.

4.3. Código final

[person.h]

```
typedef struct _person_t *person_t;

person_t person_create(unsigned int id, unsigned int age, float weight);
person_t person_destroy(person_t person);
void print_person(person_t person);
```

[person.c]

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include "person.h"

struct _person_t {
    unsigned int id;
    unsigned int age;
    float weight;
};

person_t person_create(unsigned int id, unsigned int age, float weight) {
    person_t person = NULL;

    person = calloc(1, sizeof(struct _person_t));
```

```
    assert(person != NULL);

    person->id = id;
    person->age = age;
    person->weight = weight;

    return(person);
}

person_t person_destroy(person_t person) {
    free(person);
    person = NULL;
    return(person);
}

void print_person(person_t person) {
    printf("Person %d\n", person->id);
    printf("\tage: %d, weight: %f\n", person->age, person->weight);
}
```

[main.c]

```
#include <stdlib.h>
#include "person.h"

int main(void) {
    person_t p = NULL;
    p = person_create(1, 23, 80.5);
    print_person(p);
    p = person_destroy(p);

    return(0);
}
```