

Sesión 06: Punteros

Hoja de problemas


Programación para Sistemas


Ángel Herranz

aherranz@fi.upm.es


Universidad Politécnica de Madrid


Otoño 2018

 **Ejercicio 1.** Repasa las transparencias de clase. Las transparencias de este tema están repletas de pruebas que tienes que entender a la perfección.

 **Ejercicio 2.** Poco a poco vamos viendo nueva sintaxis para las expresiones del lenguaje C. En esta sesión hemos descubierto dos sintaxis nuevas y muy importantes: `*e` y `&e`. Antes de continuar vuelve a asegurarte de que entiendes la semántica de ambas sintaxis:

- `*e`: se evalúa *e*, su resultado es entonces interpretado como una dirección de memoria, y `*e` hace referencia al contenido de dicha dirección de memoria.
- `&e`: en este caso *e* no puede ser cualquier expresión, sólo puede ser un *lvalue*¹, normalmente un identificador. El significado es «la dirección de memoria de *e*».

 **Ejercicio 3.** No olvides realizar una implementación correcta de la función intercambiar.

 **Ejercicio 4.** Ejecuta la función intercambiar bajo el control de gdb y vete explorando las variables.

¹Todo aquello que puede aparecer en la izquierda de una asignación.

Ejercicio 5.

RPN (Reverse Polish Notation)

The following algorithm evaluates postfix expressions using a stack, with the expression processed from left to right:

```
for each token in the postfix expression:
    if token is an operator:
        operand_2 = pop from the stack
        operand_1 = pop from the stack
        result = evaluate token with operand_1 and operand_2
        push result back onto the stack
    else if token is an operand:
        push token onto the stack
result = pop from the stack
```

Reverse Polish notation (Wikipedia)

En este ejercicio tendrás que programar una calculadora *polaca inversa*:


- Los *tokens* serán floats, operadores ('+', '-', '*', '/') y el caracter de *fin de expresión* ('=')
- Leemos la expresión de la entrada estándar con `scanf`²:

```
float operando;
char operador[2];
scanf("%f", &operando);
scanf("%s", operador);
```

- Utilizaremos un array para implementar la pila de operandos
- Asumiremos que la pila no puede crecer en más de 1000 elementos

Así deberá comportarse tu programa:

```
$ ./rpn
15 7 1 1 + - / 3 * 2 1 1 + + - =
5
$ |
```

-  **Ejercicio 6.** Si no lo has hecho ya, modifica tu programa `rpn` para acceder a la pila de operandos utilizando punteros.

- » **Ejercicio 7.** En este ejercicio vamos a combinar conocimiento de varios temas para crear un tipo *tipo abstracto de datos*³. El tipo abstracto de datos que tendrás que implementar será el de las pilas acotadas de enteros.

Para implementar dicho tipo abstracto de datos tendrás que usar un array de enteros, **y nada más**. Como necesitas llevar la cuenta del número de elementos que hay en la pila y la capacidad máxima, la estructura a utilizar va a ser la siguiente:

²Utiliza el manual (`man 3 scanf`), o mejor el libro K&R, apéndice B, sección 1.3.

³Aunque en C cuesta ser realmente abstracto, lo vamos a intentar.

- En la posición 0 del array guardarás la capacidad de la pila.
- En la posición 1 del array guardarás el número de elementos en la pila.
- En la posición 2 del array guardarás el primer elemento en ser apilado.
- En la posición 3 del array guardarás el segundo elemento en ser apilado.
- etc.

Vamos a construir un módulo para dicho tipo abstracto. Empezaremos con el *header*:

Listing 1: pila_acotada.h

```

1  #ifndef PILA_ACOTADA_H
2  #define PILA_ACOTADA_H
3
4  /* Inicializa una pila dejándola vacía */
5  extern void inicializar(int pila[],
6                        int capacidad);
7
8  /* Decide si la pila está vacía */
9  extern int vacia(int pila[]);
10
11 /* Decide si la pila está vacía */
12 extern int llena(int pila[]);
13
14 /* Coloca x en la cima de la pila (si la pila no está llena */
15 extern void apilar(int pila[], int x);
16
17 /* Devuelve la cima de la pila (si la pila no está vacía) */
18 extern int cima(int pila[]);
19
20 /* Elimina el elemento en la cima de la pila */
21 extern void desapilar(int pila[]);
22
23 #endif

```

Tu labor será completar e implementar correctamente las funciones en el fichero `pila_acotada.c`. Te ofrecemos aquí el esqueleto:

Listing 2: `pila_acotada.c`

```
#include "pila_acotada.h"

void inicializar(int pila[],
                int capacidad){
    /* Implementar correctamente */
}

int vacia(int pila[]) {
    /* Implementar correctamente */
    return 0;
}

int llena(int pila[]) {
    /* Implementar correctamente */
    return 0;
}

void apilar(int pila[], int x) {
    /* Implementar correctamente */
}

int cima(int pila[]) {
    /* Implementar correctamente */
    return 0;
}

void desapilar(int pila[]) {
    /* Implementar correctamente */
}
```

Para ayudarte a saber que vas por el buen camino hemos elaborado el siguiente program de test:

Listing 3: pila_acotada_test.c

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

#define MAX 1000
#define N 20

int main() {
    int pila[MAX];
    int i;

    /* Para no cagarla con los parámetros de la prueba */
    assert(N < MAX);
    assert(N > 1);

    inicializar(pila, N);

    /* Tras inicializar la pila debería estar vacía */
    assert(vacia(pila));
    /* y por tanto no llena */
    assert(!llena(pila));

    apilar(pila, 42);

    /* Tras apilar un elemento la pila no debería estar vacía */
    assert(!vacia(pila));
    /* tampoco llena (N es mayor que 1) */
    assert(!llena(pila));
    /* y la cima debería ser el 42 apilado */
    assert(cima(pila) == 42);

    desapilar(pila);

    /* Tras desapilar el único elemento la pila debería estar vacía */
    assert(vacia(pila));
    /* y no llena */
    assert(!llena(pila));

    /* Este bucle casi llena la pila */
    for (i = 1; i < N; i++) {
        apilar(pila, 2*i);
        /* Tras cada apilado la pila no debería estar vacía */
        assert(!vacia(pila));
    }
```

```

    /* tampoco llena */
    assert(!llena(pila));
    /* y la cima debería ser lo último apilado */
    assert(cima(pila) == 2*i);
}

apilar(pila, 0);

/* Tras apilar un elemento más, la pila no debería estar vacía */
assert(!vacía(pila));
/* y ahora sí, ya debería estar llena */
assert(llena(pila));
/* Y la cima debería ser lo último apilado */
assert(cima(pila) == 0);

desapilar(pila);

/* Este bucle casi vacía la pila */
for (i = N - 1; i > 1; i--) {
    /* Tras cada desapilado la pila debería tener los elementos
       apilados en orden inverso */
    assert(cima(pila) == 2*i);

    desapilar(pila);

    /* Tras cada desapilado la pila no debería estar vacía */
    assert(!vacía(pila));
    /* pero tampoco llena */
    assert(!llena(pila));
}

desapilar(pila);

/* Tras el último desapilado, la pila debería estar vacía */
assert(vacía(pila));
/* y no llena */
assert(!llena(pila));

fprintf(stderr, "¡Todos los tests pasados!\n");

return 0;
}

```

Y para que lo tengas aún más fácil, puedes utilizar el siguiente Makefile:

```
CFLAGS=-Wall -Werror -g -pedantic


pila_acotada_test: pila_acotada.o pila_acotada_test.o
    $(CC) $(CFLAGS) -o $@ $^

pila_acotada.o: pila_acotada.c pila_acotada.h

clean:
    rm -f *.o pila_acotada_test
```

Cuando todo esté correcto deberías experimentar la siguiente sesión Bash:

```
$ make
cc -Wall -Werror -g -pedantic -c -o pila_acotada_test.o pila_acotada_test.c
cc -Wall -Werror -g -pedantic -o pila_acotada_test pila_acotada.o pila_acotada_test.o
$ ./pila_acotada_test
¡Todos los tests pasados!
$ |
```

-  **Ejercicio 8.** Modifica tu implementación de *polaca inversa* `rpn` para usar tus recién estrenadas pilas acotadas.