# Soluciones - práctico 1

## Programación en C

### Ejercicio 6

a) La función **func** recibe una cadena, y retorna esta misma habiéndole modificado los caracteres en minúscula por mayúscula.

b) El indicador de formato **"%[^\n]"** se utiliza para leer hasta que se teclee un "\n" o fin de linea (es decir, un <Enter>). Más genericamente dentro de los corchetees se colocan los caracteres que serán aceptados para leer: **"%[abcdef]"** aceptaría únicamente los caracteres desde la **a** hasta la **f**. Cuando se le agrega **"^"** al principio del conjunto de caracteres se invierte el sentido del indicador, este pasa a querer decir que se aceptarán todos los caracteres que no estén dentro del conjunto especificado.

c) Simplemente que se evalúa una única vez el largo del string con la función **strlen** (esta recorre todo el arreglo para conocer su tamaño).

Si la sentencia fuera:

```
for (i = 0; i < strlen(s); i++)
```

Se evaluaría **strlen (s) + 1** veces el largo del string.

d) El largo del string ingresado desde la linea de comandos está restringido al valor de la constante **MAXIMO_DE_LINEA**.

### Ejercicio 7 (*)

Indique que efecto tiene cada uno de los siguientes trozos de código:

**a) int c[5];**

Declara un arreglo constante de enteros de largo 5.

**b) int c[5] = {0};**

Declara un arreglo constante de enteros de largo 5 e inicializa todas las posiciones en 0.

**c) int n[5] = {10, 20, 30, 40, 50};**

Declara un arreglo constante de enteros de largo 5 e inicializa las posiciones con los valores especificados.

**d) int n[] = {4, 3, 2, 1, 0};**

Declara un arreglo constante de enteros de largo determinado por la cantidad de valores especificados e inicializa las posiciones con estos valores.

**e) char c[] = “hola”;**

Define un arreglo de tamaño dado por el largo del string **"hola"** + el carácter de fin de string. Los caracteres que componen este arreglo pueden ser modificados.

**f) int d[3][3] = {{1, 2, 3}, {2, 3, 4}, {5, 6, 7}};**

Define una matriz de tamaño 3x3 inicializada con los valores especificados.

**k) char* c = “puntero_a_char”;**

Define un arreglo constante de tamaño dado por el largo del string **"puntero_a_char"** + el caracter de fin de string. Los caracteres que compone este arreglo no pueden ser modificados.

**l) int c[3] = {‘1’,’2’,’3’};**

Se inicializa con los valores ASCII de los caracteres.

```
int *p = c;
cout << *p;
```

Imprime el valor ASCII del primer caracter definido

```
cout << p[0];
```

Imprime el valor ASCII del primer caracter definido

```
cout << p[1];
```

Imprime el valor ASCII del segundo caracter definido

```
p++;
cout << p[1];
```

Imprime el valor ASCII del tercer caracter definido

**n)**

```
int a = 1500;
int b = 2000;
int c = 3000;

int **doble_puntero;
doble_puntero = new int*[3];
```

Define un vector de punteros a valores enteros, reservando memoria para tres elementos.**doble_puntero[0] = &a;**

```
doble_puntero[1] = &b;
doble_puntero[2] = &c;
```

Define cada posición del arreglo como una referencia a **a,b** y **c**

```
for(int i = 0; i <= 2; i++)
  cout << *(doble_puntero[i]);
```

Imprime la información de **a,b** y **c**.

## EJERCICIO 8 (*)

Explique cuáles son los errores de los siguientes trozos de código:

**a)**
```
int a[3];
a = new int[10];
```

Intenta reservar memoria para un vector definido estáticamente. No compila.

**b)**
```
int a[3] = {1, 2, 3, 4};
```

Asigna más posiciones de las que hay reservadas en la definición del vector. No compila.

**c)**
```
char* c = new char[10];
for (int i = 1; i <= 10; i++)
  c[i] = i;
```

Intenta inicializar una posición de memoria no reservada.

**d)**
```
struct s{int a, b;};
s* puntero_a_s = new s;
*puntero_a_s.a = 1;
```

El operador **(.)** tiene mayor precedencia que el **(*)**. Compila y da error al ejecutar.

```
puntero_a_s->b = 1;
(*puntero_a_s).a = 1;
```

**e)**
```
#define TAMANIO_MATRIZ = 10
int** matriz;
matriz = (int*) malloc (sizeof (int) * TAMANIO_MATRIZ);
```

El = en la definición de la macro está mal sintácticamente.

Se reserva memoria de tipo **int** cuando se debería pedir memoria de tipo **int***, a su vez el casting debe ser de tipo **int****.

**f)**
```
int numero = 5;
int* p_numero = numero;
```

Se asigna un valor entero a un puntero a un entero. No compila.

**g)**
```
char* p_char = "pala";
p_char[2] = 't';
```

Se intenta modificar un vector inicializado con valores estáticos.

**h)**
```
int* func ()
{
      int i = 5;
      return &i;
}

void main ()
{
      int* p_i = func ();
      printf ("%d", *p_i);
}
```

Cuando se sale de la función **func**, la posición de memoria reservada para **i** desaparece, por lo tanto se provoca un error al momento de imprimir el valor desreferenciando el puntero **p_i**.

**i)**
```
int a, b;

cout << "\nPor favor, ingrese un número: ";
cin >> a;
cout << "\nIngrese otro número: ";
cin >> b;

if (a = b) cout << "\n Son iguales";
else cout << "Son distintos";
```

**(a = b)** es una asignación que retorna **false** cuando el valor de **b** es igual a 0, y **true** en caso contrario. Para comparar si estas variables contienen el mismo valor se debería utilizar la sentencia **(a == b)**.

**j)**
```
void darValor (int val, int* ptr)
{
      int* aux = new int;
      *aux = val;
      ptr = aux;
}

void main ()
{
      int* ptr;
      darValor (3, ptr);
      cout << *ptr;
}
```

Dentro de la función **darValor** se modifica el valor de un puntero que esta pasado por copia, por lo tanto la información de **ptr** dentro de la función **main** NO será modificada. Por esto, la memoria pedida en la función interna se pierde. Si en el encabezado de la función hubiera un & entonces estaría bien.

### EJERCICIO 14

```
#include <math.h>
#include <iostream>
#define MAX_VAL 50
using namespace std;

int main()
{
       for (int i = 1; i <= MAX_VAL; i++)
          for (int j = 1; j <= MAX_VAL; j++)
             for (int k = 2; k <= MAX_VAL; k++)
                if (pow (i, 2) + pow (j, 2) == pow(k, 2))
                     cout << "(" << i << "," << j << "," << k << ")" << endl;
}
```

### EJERCICIO 21

### (Parte b) Definición de Pila de enteros - Representación acotada

```
typedef struct NodoPilaEnt * PilaEnt;

// Constructoras

// Construye la pila vacia de capacidad tamMax.
void PilaCrear (PilaEnt & p, int tamMax);

// Agrega el entero n a la pila.
// Precondicion: La pila no está llena.
PilaEnt PilaInsertar (PilaEnt p, int n);

// Predicados

// Retorna true si la pila está vacía y false en caso contrario.
bool PilaEstaVacia(PilaEnt p);

// Retorna true si la pila está llena y false en caso contrario.
bool PilaEstaLlena(PilaEnt p);

// Selectoras

// Saca el tope de la pila.
// Precondicion: La pila no está vacía.
PilaEnt PilaSacar (PilaEnt p);

// Retorna el tope de la pila.
// Precondicion: La pila no está vacía.
int PilaTope (PilaEnt p);

// Destructoras

// Libera la memoria de la pila.
void PilaDestruir (PilaEnt & p);
```

## Implementación de Pila de enteros - Representación acotada.

```
#include "PilaAcotada.h"

struct NodoPilaEnt
{
        int tope;    // indica la posición del último elemento la pila
        int tamanio; // indica la cantidad máxima de elementos de la pila
        int * elementos;
};

void PilaCrear(PilaEnt &p, int tamMax)
{
   p           = new (struct NodoPilaEnt);
   p->tope     = -1;
   p->tamanio  = tamMax;
   p->elementos = new int [tamMax];
}

PilaEnt PilaInsertar(PilaEnt p, int n)
{
   p->tope = p->tope + 1;
   p->elementos[p->tope] = n;
   return p;
}

bool PilaEstaVacia(PilaEnt p)
{
   return (p->tope == -1);
}

bool PilaEstaLlena(PilaEnt p)
{
   return (p->tope == (p->tamanio - 1));
}

PilaEnt PilaSacar(PilaEnt p)
{
   p->tope = p->tope - 1;
   return p;
}

int PilaTope(PilaEnt p)
{
   return (p->elementos[p->tope]);
}

void PilaDestruir (PilaEnt & p)
{
   delete [] p->elementos;
   delete p;
}
```

## Definición de Pila de enteros - Representación no acotada

Notar que **PilaEnt** ahora es un nodo y las operación reciben punteros a **PilaEnt**. Observar además (en el módulo de implementación) el chequeo de las precondiciones utilizando la función `assert`.

```
struct PilaEnt;

// Constructoras

// Construye la pila vacia
PilaEnt* PilaCrear();

// Agrega el entero n a la pila.
void PilaInsertar(PilaEnt* p, int n);

// Predicados

// Retorna true si la pila está vacía y false en caso contrario.
bool PilaEstaVacia(PilaEnt* p);

// Selectoras

// Saca el tope de la pila.
// Precondicion: La pila no está vacía.
void PilaSacar (PilaEnt* p);

// Retorna el tope de la pila.
// Precondicion: La pila no está vacía.
int PilaTope (PilaEnt* p);

// Destructoras

// Libera la memoria de la pila.
void PilaDestruir (PilaEnt* p);
```

**Implementación de Pila de enteros - Representación no acotada.**

```
#include "PilaNoAcotada.h"
#include <assert.h>
#include <stddef.h> // macro NULL

struct Nodo
{
    int n;
    Nodo* next;
};

struct PilaEnt
{
    Nodo* head;
};

PilaEnt* PilaCrear()
{
    PilaEnt* res = new PilaEnt;
    res->head = NULL;
    return res;
}

void PilaInsertar(PilaEnt* p, int n)
{
    Nodo* head = new Nodo;
    head->n = n;
    head->next = p->head;
    p->head = head;
}

bool PilaEstaVacia(PilaEnt* p)
{
    return NULL == p->head;
}

void PilaSacar(PilaEnt* p)
{
    assert(!PilaEstaVacia(p));
    Nodo* sobra = p->head;
    p->head = sobra->next;
    delete sobra;
}

int PilaTope(PilaEnt* p)
{
    assert(!PilaEstaVacia(p));
    return p->head->n;
}

void PilaDestruir (PilaEnt* p)
{
    while(!PilaEstaVacia(p))
    {
        PilaSacar(p);
    }
    delete p;
}
```