Explicación de la práctica 3

Arreglos, punteros y strings

Seminario de Lenguajes opción C

Facultad de Informática
Universidad Nacional de La Plata

2017

Indice

Punteros: Repaso

- Declaración:

```
tipo *nombre;
```

- Asignación:

```
int variable;
int *puntero = &variable;
```

- Dereferenciación (obtener el valor):

```
int otra = *puntero;
```

Punteros: Repaso

- Declaración:

```
tipo *nombre;
```

- Asignación:

```
int variable;
int *puntero = &variable;
```

- Dereferenciación (obtener el valor):

```
int otra = *puntero;
```

Punteros: Repaso

- Declaración:

```
tipo *nombre;
```

- Asignación:

```
int variable;
int *puntero = &variable;
```

- Dereferenciación (obtener el valor):

```
int otra = *puntero;
```

void

- En la lista de argumentos:

```c
int funcion(void) // No recibe argumentos
```

- Como valor de retorno:

```c
void funcion(int x) // No retorna ningún valor
```

- Como puntero:

```c
int variable;
void *puntero; // Un puntero genérico (sin tipo)
puntero = &variable;
```

void

- En la lista de argumentos:

```
int funcion(void) // No recibe argumentos
```

- Como valor de retorno:

```
void funcion(int x) // No retorna ningún valor
```

- Como puntero:

```
int variable;
void *puntero; // Un puntero genérico (sin tipo)
puntero = &variable;
```

void

- En la lista de argumentos:

```
int funcion(void) // No recibe argumentos
```

- Como valor de retorno:

```
void funcion(int x) // No retorna ningún valor
```

- Como puntero:

```
int variable;
void *puntero; // Un puntero genérico (sin tipo)
puntero = &variable;
```

Usos de void*

- Manipular punteros sin saber el tipo (como hacen malloc, calloc, free, etc...).
- Estructuras de datos genéricas.
  - Vectores
  - Listas
  - Conjuntos
  - Etc...
- Para desreferenciarlo debemos conocer su tipo:

```
int valor1 = 5;
int valor2;
void *puntero = &valor1;
...
valor2 = *((int *) puntero);
```

Usos de void*

- Manipular punteros sin saber el tipo (como hacen malloc, calloc, free, etc...).
- Estructuras de datos genéricas.
  - Vectores
  - Listas
  - Conjuntos
  - Etc...
- Para desreferenciarlo debemos conocer su tipo:

```
int valor1 = 5;
int valor2;
void *puntero = &valor1;
...
valor2 = *((int *) puntero);
```

Usos de void*

- Manipular punteros sin saber el tipo (como hacen malloc, calloc, free, etc...).
- Estructuras de datos genéricas.
  - Vectores
  - Listas
  - Conjuntos
  - Etc...
- Para desreferenciarlo debemos conocer su tipo:

```
int valor1 = 5;
int valor2;
void *puntero = &valor1;
...
valor2 = *((int *) puntero);
```

Usos de void*

- Manipular punteros sin saber el tipo (como hacen malloc, calloc, free, etc...).
- Estructuras de datos genéricas.
  - Vectores
  - Listas
  - Conjuntos
  - Etc...
- Para desreferenciarlo debemos conocer su tipo:

```
int valor1 = 5;
int valor2;
void *puntero = &valor1;
...
valor2 = *((int *) puntero);
```

Arreglos

- Definimos arreglos indicando su tipo y tamaño:

```
int arreglo[5];
int matriz[10][20];
```

- Si los usamos sin corchetes obtenemos un puntero al primer elemento:

```
arreglo == &arreglo[0]
```

- Pero `sizeof()` lo sigue tratando como arreglo:

```
sizeof(arreglo) == 5 * sizeof(int)
```

- A `sizeof()` le importa el tipo del parámetro únicamente:

```
int *x = arreglo;
sizeof(x) == 4 // en una máquina de 32 bits
```

Arreglos

- Definimos arreglos indicando su tipo y tamaño:

```
int arreglo[5];
int matriz[10][20];
```

- Si los usamos sin corchetes obtenemos un puntero al primer elemento:

```
arreglo == &arreglo[0]
```

- Pero `sizeof()` lo sigue tratando como arreglo:

```
sizeof(arreglo) == 5 * sizeof(int)
```

- A `sizeof()` le importa el tipo del parámetro únicamente:

```
int *x = arreglo;
sizeof(x) == 4 // en una máquina de 32 bits
```

Arreglos

- Definimos arreglos indicando su tipo y tamaño:

```
int arreglo[5];
int matriz[10][20];
```

- Si los usamos sin corchetes obtenemos un puntero al primer elemento:

```
arreglo == &arreglo[0]
```

- Pero `sizeof()` lo sigue tratando como arreglo:

```
sizeof(arreglo) == 5 * sizeof(int)
```

- A `sizeof()` le importa el tipo del parámetro únicamente:

```
int *x = arreglo;
sizeof(x) == 4 // en una máquina de 32 bits
```

Arreglos

- Definimos arreglos indicando su tipo y tamaño:

```
int arreglo[5];
int matriz[10][20];
```

- Si los usamos sin corchetes obtenemos un puntero al primer elemento:

```
arreglo == &arreglo[0]
```

- Pero `sizeof()` lo sigue tratando como arreglo:

```
sizeof(arreglo) == 5 * sizeof(int)
```

- A `sizeof()` le importa el tipo del parámetro únicamente:

```
int *x = arreglo;
sizeof(x) == 4 // en una máquina de 32 bits
```

Arreglos y funciones

- Las funciones NO reciben arreglos como argumentos.
- En cambio reciben un puntero (verificar con `sizeof()`).
- Las siguientes declaraciones son equivalentes:

```
int funcion(int a[]);
int funcion(int a[5]); // El tamaño se descarta
int funcion(int *a);
```

Arreglos y funciones

- Las funciones NO reciben arreglos como argumentos.
- En cambio reciben un puntero (verificar con `sizeof()`).
- Las siguientes declaraciones son equivalentes:

```
int funcion(int a[]);
int funcion(int a[5]); // El tamaño se descarta
int funcion(int *a);
```

Arreglos y funciones

- Las funciones NO reciben arreglos como argumentos.
- En cambio reciben un puntero (verificar con `sizeof()`).
- Las siguientes declaraciones son equivalentes:

```
int funcion(int a[]);
int funcion(int a[5]); // El tamaño se descarta
int funcion(int *a);
```

Si es un parámetro, entonces no es un arreglo...


```c
#include <stdio.h>
int funcion(int a[], int b[5], int *c){
    printf("a = %d, b = %d, c = %d\n",
           sizeof(a), sizeof(b), sizeof(c));
    return 0;
}
int main(){
    int x[25];
    int y[20];
    int z[10];
    printf("x = %d, y = %d, z = %d\n",
           sizeof(x), sizeof(y), sizeof(z));
    funcion(x, y, z);
    return 0;
}
```

Imprime:

```
x = 100, y = 80, z = 40
a = 4, b = 4, c = 4
```

Si es un parámetro, entonces no es un arreglo...


```c
#include <stdio.h>
int funcion(int a[], int b[5], int *c){
    printf("a = %d, b = %d, c = %d\n",
           sizeof(a), sizeof(b), sizeof(c));
    return 0;
}
int main(){
    int x[25];
    int y[20];
    int z[10];
    printf("x = %d, y = %d, z = %d\n",
           sizeof(x), sizeof(y), sizeof(z));
    funcion(x, y, z);
    return 0;
}
```

Imprime:

```
x = 100, y = 80, z = 40
a = 4, b = 4, c = 4
```

Arreglos multidimensionales

- Al declararlo se indican el tipo de los elementos y las dimensiones:

```
int matriz[4][3];
```

- Si bien es la abstracción de una matriz y la pensamos como:

f0	f0	f0
f1	f1	f1
f2	f2	f2
f3	f3	f3

- Se almacenan en memoria por fila, primero la fila 0, luego la 1, etc...

f0	f0	f0	f1	f1	f1	f2	f2	f2	f3	f3	f3

Arreglos multidimensionales

- Al declararlo se indican el tipo de los elementos y las dimensiones:

```
int matriz[4][3];
```

- Si bien es la abstracción de una matriz y la pensamos como:

f0	f0	f0
f1	f1	f1
f2	f2	f2
f3	f3	f3

- Se almacenan en memoria por fila, primero la fila 0, luego la 1, etc...

f0	f0	f0	f1	f1	f1	f2	f2	f2	f3	f3	f3

Arreglos multidimensionales

- Al declararlo se indican el tipo de los elementos y las dimensiones:

```
int matriz[4][3];
```

- Si bien es la abstracción de una matriz y la pensamos como:

f0	f0	f0
f1	f1	f1
f2	f2	f2
f3	f3	f3

- Se almacenan en memoria por fila, primero la fila 0, luego la 1, etc...

f0	f0	f0	f1	f1	f1	f2	f2	f2	f3	f3	f3

Matrices y funciones

- Al igual que con los arreglos de 1 dimensión, se pasa un puntero.
- Pero hay que especificar la geometría (al menos las columnas).

```
int function(int matriz[][20]){
    printf("%d\n", matriz[3][5]);
}
```

- Si quisieramos simular eso sin usar corchetes:

```
int function(int *matriz){
    printf("%d\n", *(matriz + 20 * 3 + 5));
}
// La invocamos: function((int *) matriz);
```

Matrices y funciones

- Al igual que con los arreglos de 1 dimensión, se pasa un puntero.
- Pero hay que especificar la geometría (al menos las columnas).

```
int function(int matriz[][20]){
    printf("%d\n", matriz[3][5]);
}
```

- Si quisieramos simular eso sin usar corchetes:

```
int function(int *matriz){
    printf("%d\n", *(matriz + 20 * 3 + 5));
}
// La invocamos: function((int *) matriz);
```

Matrices y funciones

- Al igual que con los arreglos de 1 dimensión, se pasa un puntero.
- Pero hay que especificar la geometría (al menos las columnas).

```
int function(int matriz[][20]){
    printf("%d\n", matriz[3][5]);
}
```

- Si quisieramos simular eso sin usar corchetes:

```
int function(int *matriz){
    printf("%d\n", *(matriz + 20 * 3 + 5));
}
// La invocamos: function((int *) matriz);
```

Strings

- Son arreglos de char terminados en 0.

```
char x[] = {'H', 'o', 'l', 'a', 0};
char y[5] = "Hola";
```

- No se pueden comparar con ==.
- No se copian con = (salvo en la declaración de un arreglo).
- Los strings literales son de solo lectura:

```
char *mensaje = "Hola"; // Apunta a "Hola"
char copia[] = "Hola"; // Es una copia de "Hola"
mensaje[0] = 64; // Es un error
copia[0] = 64; // Es correcto
```

Strings

- Son arreglos de char terminados en 0.

```
char x[] = {'H', 'o', 'l', 'a', 0};
char y[5] = "Hola";
```

- No se pueden comparar con ==.
- No se copian con = (salvo en la declaración de un arreglo).
- Los strings literales son de solo lectura:

```
char *mensaje = "Hola"; // Apunta a "Hola"
char copia[] = "Hola"; // Es una copia de "Hola"
mensaje[0] = 64; // Es un error
copia[0] = 64; // Es correcto
```

Strings

- Son arreglos de char terminados en 0.

```
char x[] = {'H', 'o', 'l', 'a', 0};
char y[5] = "Hola";
```

- No se pueden comparar con ==.
- No se copian con = (salvo en la declaración de un arreglo).
- Los strings literales son de solo lectura:

```
char *mensaje = "Hola"; // Apunta a "Hola"
char copia[] = "Hola"; // Es una copia de "Hola"
mensaje[0] = 64; // Es un error
copia[0] = 64; // Es correcto
```

Strings

- Son arreglos de char terminados en 0.

```
char x[] = {'H', 'o', 'l', 'a', 0};
char y[5] = "Hola";
```

- No se pueden comparar con ==.
- No se copian con = (salvo en la declaración de un arreglo).
- Los strings literales son de solo lectura:

```
char *mensaje = "Hola"; // Apunta a "Hola"
char copia[] = "Hola"; // Es una copia de "Hola"
mensaje[0] = 64; // Es un error
copia[0] = 64; // Es correcto
```

Strings

- Son arreglos de char terminados en 0.

```
char x[] = {'H', 'o', 'l', 'a', 0};
char y[5] = "Hola";
```

- No se pueden comparar con ==.
- No se copian con = (salvo en la declaración de un arreglo).
- Los strings literales son de solo lectura:

```
char *mensaje = "Hola"; // Apunta a "Hola"
char copia[] = "Hola"; // Es una copia de "Hola"
mensaje[0] = 64; // Es un error
copia[0] = 64; // Es correcto
```

Strings

- Son arreglos de char terminados en 0.

```
char x[] = {'H', 'o', 'l', 'a', 0};
char y[5] = "Hola";
```

- No se pueden comparar con ==.
- No se copian con = (salvo en la declaración de un arreglo).
- Los strings literales son de solo lectura:

```
char *mensaje = "Hola"; // Apunta a "Hola"
char copia[] = "Hola"; // Es una copia de "Hola"
mensaje[0] = 64; // Es un error
copia[0] = 64; // Es correcto
```

Funciones de strings

- strcmp()
- strcpy()
- strcat()
- strlen()
- sscanf()
- sprintf()
- fgets()

Las primeras 4 declaradas en `string.h`.

Funciones de strings

- strcmp()
- strcpy()
- strcat()
- strlen()
- sscanf()
- sprintf()
- fgets()

Las primeras 4 declaradas en `string.h`.

Funciones de strings

- strcmp()
- strcpy()
- strcat()
- strlen()
- sscanf()
- sprintf()
- fgets()

Las primeras 4 declaradas en `string.h`.

Funciones de strings

- strcmp()
- strcpy()
- strcat()
- strlen()
- sscanf()
- sprintf()
- fgets()

Las primeras 4 declaradas en `string.h`.

Funciones de strings

- strcmp()
- strcpy()
- strcat()
- strlen()
- sscanf()
- sprintf()
- fgets()

Las primeras 4 declaradas en `string.h`.

Funciones de strings

- strcmp()
- strcpy()
- strcat()
- strlen()
- sscanf()
- sprintf()
- fgets()

Las primeras 4 declaradas en `string.h`.

Funciones de strings

- strcmp()
- strcpy()
- strcat()
- strlen()
- sscanf()
- sprintf()
- fgets()

Las primeras 4 declaradas en `string.h`.

Argumentos del programa

- Cuando invocamos a gcc le pasamos argumentos como `-Wall`, `-o`, los nombres de los archivos, etc...
- Cualquier programa puede recibir argumentos.
- En C los leemos con argc y argv:
  - **argc:** Cantidad de parámetros (al menos 1, el nombre del programa).
  - **argv:** "Arreglo" de strings, cada uno es uno de los parámetros en orden.
- Por ejemplo:

```c
int main(int argc, char *argv[]){
    int i;
    for (i = 0; i < argc; i++){
        puts(argv[i]);
    }
    return 0;
}
```

Argumentos del programa

- Cuando invocamos a gcc le pasamos argumentos como `-Wall`, `-o`, los nombres de los archivos, etc...
- Cualquier programa puede recibir argumentos.
- En C los leemos con argc y argv:
  - **argc:** Cantidad de parámetros (al menos 1, el nombre del programa).
  - **argv:** "Arreglo" de strings, cada uno es uno de los parámetros en orden.
- Por ejemplo:

```c
int main(int argc, char *argv[]){
    int i;
    for (i = 0; i < argc; i++){
        puts(argv[i]);
    }
    return 0;
}
```

Argumentos del programa

- Cuando invocamos a gcc le pasamos argumentos como `-Wall`, `-o`, los nombres de los archivos, etc...
- Cualquier programa puede recibir argumentos.
- En C los leemos con argc y argv:
  - **argc:** Cantidad de parámetros (al menos 1, el nombre del programa).
  - **argv:** "Arreglo" de strings, cada uno es uno de los parámetros en orden.
- Por ejemplo:

```c
int main(int argc, char *argv[]){
    int i;
    for (i = 0; i < argc; i++){
        puts(argv[i]);
    }
    return 0;
}
```

Argumentos del programa

- Cuando invocamos a gcc le pasamos argumentos como `-Wall`, `-o`, los nombres de los archivos, etc...
- Cualquier programa puede recibir argumentos.
- En C los leemos con argc y argv:
  - **argc:** Cantidad de parámetros (al menos 1, el nombre del programa).
  - **argv:** "Arreglo" de strings, cada uno es uno de los parámetros en orden.
- Por ejemplo:

```c
int main(int argc, char *argv[]){
    int i;
    for (i = 0; i < argc; i++){
        puts(argv[i]);
    }
    return 0;
}
```

Argumentos del programa

- Cuando invocamos a gcc le pasamos argumentos como `-Wall`, `-o`, los nombres de los archivos, etc...
- Cualquier programa puede recibir argumentos.
- En C los leemos con argc y argv:
  - **argc:** Cantidad de parámetros (al menos 1, el nombre del programa).
  - **argv:** "Arreglo" de strings, cada uno es uno de los parámetros en orden.
- Por ejemplo:

```
int main(int argc, char *argv[]){
    int i;
    for (i = 0; i < argc; i++){
        puts(argv[i]);
    }
    return 0;
}
```

Argumentos del programa

- Cuando invocamos a gcc le pasamos argumentos como `-Wall`, `-o`, los nombres de los archivos, etc...
- Cualquier programa puede recibir argumentos.
- En C los leemos con argc y argv:
  - **argc:** Cantidad de parámetros (al menos 1, el nombre del programa).
  - **argv:** "Arreglo" de strings, cada uno es uno de los parámetros en orden.
- Por ejemplo:

```c
int main(int argc, char *argv[]){
    int i;
    for (i = 0; i < argc; i++){
        puts(argv[i]);
    }
    return 0;
}
```

Alocación de memoria

- `void *malloc(size_t)`: Recibe un tamaño en bytes, retorna un puntero a la memoria alocada.
- `void free(void *)`: Recibe un puntero a memoria alocada con `malloc` (u otro) y la libera.
- `void *calloc(size_t nmemb, size_t size)`: Aloca (nmemb * size) bytes inicializados en cero.
- `NULL`: Es una macro que representa una dirección de memoria imposible. Se usa para inicializar punteros.
- Recordar liberar todo lo que se aloca.

Alocación de memoria

- `void *malloc(size_t)`: Recibe un tamaño en bytes, retorna un puntero a la memoria alocada.
- `void free(void *)`: Recibe un puntero a memoria alocada con `malloc` (u otro) y la libera.
- `void *calloc(size_t nmemb, size_t size)`: Aloca (nmemb * size) bytes inicializados en cero.
- `NULL`: Es una macro que representa una dirección de memoria imposible. Se usa para inicializar punteros.
- Recordar liberar todo lo que se aloca.

Alocación de memoria

- `void *malloc(size_t)`: Recibe un tamaño en bytes, retorna un puntero a la memoria alocada.
- `void free(void *)`: Recibe un puntero a memoria alocada con `malloc` (u otro) y la libera.
- `void *calloc(size_t nmemb, size_t size)`: Aloca (nmemb * size) bytes inicializados en cero.
- `NULL`: Es una macro que representa una dirección de memoria imposible. Se usa para inicializar punteros.
- Recordar liberar todo lo que se aloca.

Alocación de memoria

- `void *malloc(size_t)`: Recibe un tamaño en bytes, retorna un puntero a la memoria alocada.
- `void free(void *)`: Recibe un puntero a memoria alocada con `malloc` (u otro) y la libera.
- `void *calloc(size_t nmemb, size_t size)`: Aloca (nmemb * size) bytes inicializados en cero.
- `NULL`: Es una macro que representa una dirección de memoria imposible. Se usa para inicializar punteros.
- Recordar liberar todo lo que se aloca.

Alocación de memoria

- `void *malloc(size_t)`: Recibe un tamaño en bytes, retorna un puntero a la memoria alocada.
- `void free(void *)`: Recibe un puntero a memoria alocada con `malloc` (u otro) y la libera.
- `void *calloc(size_t nmemb, size_t size)`: Aloca (nmemb * size) bytes inicializados en cero.
- `NULL`: Es una macro que representa una dirección de memoria imposible. Se usa para inicializar punteros.
- Recordar liberar todo lo que se aloca.

Ejercicios

1. Ver ejercicio extra de la práctica.
2. Desarrollar una pila de enteros, usando un arreglo y una variable tope.
3. Permitir variar el tamaño de la pila con el argumento `--size` al programa (usar memoria dinámica).
4. Hacer una función que dada una pila de enteros y un string guarde en este último los números separados por coma.

Ejercicios

1. Ver ejercicio extra de la práctica.
2. Desarrollar una pila de enteros, usando un arreglo y una variable tope.
3. Permitir variar el tamaño de la pila con el argumento `--size` al programa (usar memoria dinámica).
4. Hacer una función que dada una pila de enteros y un string guarde en este último los números separados por coma.

Ejercicios

1. Ver ejercicio extra de la práctica.
2. Desarrollar una pila de enteros, usando un arreglo y una variable tope.
3. Permitir variar el tamaño de la pila con el argumento `--size` al programa (usar memoria dinámica).
4. Hacer una función que dada una pila de enteros y un string guarde en este último los números separados por coma.

Ejercicios

1. Ver ejercicio extra de la práctica.
2. Desarrollar una pila de enteros, usando un arreglo y una variable tope.
3. Permitir variar el tamaño de la pila con el argumento `--size` al programa (usar memoria dinámica).
4. Hacer una función que dada una pila de enteros y un string guarde en este último los números separados por coma.