Practica en C de FSO

Para esta practica, hemos dedicado tiempo a leer las trasparencias facilitadas en la asignatura, y además de la página web que también esta en aulas.

Lo primero, creamos un repositorio en git, y lo enlazamos con eclipse para facilitar la cooperación a la hora de hacer la práctica.

Si es de interés, el repositorio es el siguiente: <https://gitlab.inf.uva.es/carmuno/proyectosc.git> (tiene los permisos necesarios para poder clonarlo).

En la primera propuesta de ejercicio se nos pide comentar el uso de la función **sizeof()**. En C hemos visto que es muy importante el tema de reservar memoria, y cuando declaramos una estructura de datos como por ejemplo un array, debemos de decir cuánto va a ocupar en memoria la estructura de datos que hemos creado, si nos pasamos, accedemos a otras partes de memoria con lo que se podría pisar nuestro propio programa o borrar cosas que no queremos al sobrescribir.

Para utilizar esta función se debe meter como argumento una estructura de datos y se obtendrá como resultado la longitud en bytes que ocupa la misma, esto nos da una pista sobre el tamaño que deberíamos reservar en memoria para esa estructura.

Si queremos mostrar el tamaño de la estructura utilizamos la función “printf”

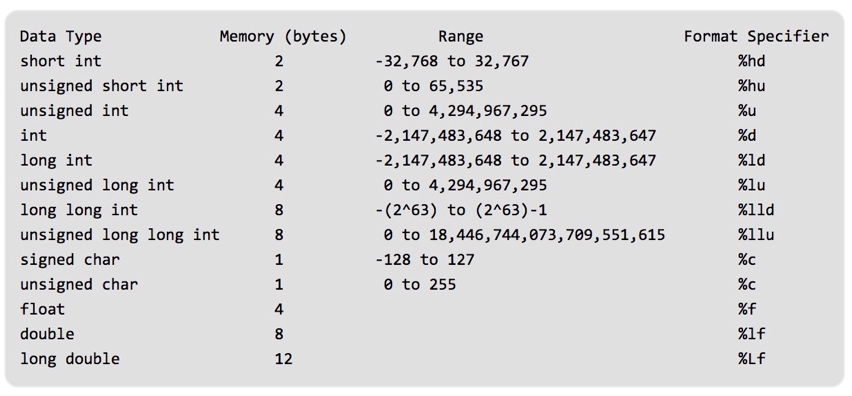
printf ("%d\n", sizeof (estructura definida)); /\* imprime el tamaño que ocupa para la estructura. Y como apunte, imprime el tamaño en decimal por el %d.

printf ("%d\n", sizeof(a));

printf ("%d\n", sizeof(b));

printf ("%d\n", sizeof(c));

Para obtener el valor máximo de las variables hemos utilizado valores en hexadecimal, mediante los cuales hemos obtenido unos resultados que concuerdan con lo encontrado en internet sobre los rangos máximos de los tipos de datos.



Respecto a los aumentos y decrementos de los iteradores, hemos visto que no es igual que en java. Hemos consultado alguna página para poder comprender este concepto. Si es de interés consultar:

<https://stackoverflow.com/questions/24853/c-what-is-the-difference-between-i-and-i>

A diferencia de java, los “++”, y los “--” se pueden colocar delante de la variable, teniendo un significado diferente.

- Si ponemos i++ aumentamos el iterador pero retornamos el valor anterior de i antes de ser incrementado.

- Si ponemos ++i aumentamos el iterador y retornamos el valor incrementado.

Los operadores en C son muy parecidos a los de java. Dejamos un ejemplo de los mismos:

|  |  |
| --- | --- |
| **Operador** | **Significado** |
| **Operador <** | Menor que |
| **Operador <=** | Menor o igual que |
| **Operador >** | Mayor que |
| **Operador >=** | Mayor o igual que |
| **Operador ==** | Igual a |
| **Operador !=** | Distinto de ó no igual que |

Análogamente como lo hacíamos en java:

|  |  |
| --- | --- |
| **OPERADOR** | **DESCRIPCIÓN** |
| **==** | Es igual |
| **!=** | Es distinto |
| **<, <=, >, >=** | Menor, menor o igual, mayor, mayor o igual |
| **&&** | Operador and (y) |
| **||** | Operador or (o) |
| **!** | Operador not (no) |

**while** (a <= 5 || a > 3)

En este caso el valor devuelto será de verdadero o falso ya que estamos tratando con operadores lógicos cuyo único retorno es de verdadero o falso.

Respecto a los operadores binarios, se hace parecido a java (nunca hemos operado con operadores binarios en java).

Para Java y C:

|  |  |  |
| --- | --- | --- |
| << | A << B | Desplazamiento de A a la izquierda en B posiciones |
| >> | A >> B | Desplazamiento de A a la derecha en B posiciones, tiene en cuenta el signo. |
| >>> | A >>> B | Desplazamiento de A a la derecha en B posiciones, no tiene en cuenta el signo. |
| & | A & B | Operación AND a nivel de bits |
| | | A | B | Operación OR a nivel de bits |
| ^ | A ^ B | Operación XOR a nivel de bits |
| ~ | ~A | Complemento de A a nivel de bits |

Una vez entendido el funcionamiento del Sizeof y los diferentes operadores básicos de C, hemos creado un pequeño programa que calcula el tamaño de cada uno de los tipos de datos .

**#include** <stdio.h>

**#define** **NAME\_LENGTH** 10

**#define** **TABLE\_SIZE** 100

**#define** **UNITS\_NUMBER** 10

**struct** **unit** { /\* Define a struct with an internal union \*/

**int** x;

**float** y;

**double** z;

**short** **int** a;

**long** b;

**union** { /\* Union with no name because it is internal to the struct \*/

**char** name[NAME\_LENGTH];

**int** id;

**short** **int** sid;

} identifier;

};

**int** **main**(**int** argc, **char** \*argv[]) {

**int** table[TABLE\_SIZE];

**struct** **unit** data[UNITS\_NUMBER];

**printf**("%d\n", **sizeof**(**struct** **unit**)); /\* Print size of structure \*/

**printf**("%d\n", **sizeof**(table)); /\* Print size of table of ints \*/

**printf**("%d\n", **sizeof**(data)); /\* Print size of table of structs \*/

**return** 0;

}