Centro Universitario de Ciencias Exactas e Ingenierías

Actividad 03 – Tipado en C

Pérez Tovar Santiago

Programación de sistemas avanzados
Sección D01

Fecha de Entrega: 26e/Febrero /2024

Objetivo:

Entender las características del tipado débil en C mediante ejemplos prácticos y reflexionar sobre sus implicaciones.

Parte 1. Ejercicio de programación:

Código 1: Conversión implícita de tipos

```c
#include <stdio.h>
int main(){
    int entero = 10;
    float flotante = 5.5;
    float resultado = entero + flotante; // Conversión implícita de int a float
    printf("Resultado: %f\n", resultado);
    return 0;
}
```

Preguntas para reflexión:

¿Qué tipo de conversión ocurre en la línea float resultado = entero + flotante?

Respuesta: La variable “**resultado”** al estar declarada como tipo **float**, provoca que las variables que la generan también se conviertan de manera implícita en **float.** Esto haciendo que el **“Entero”** pase de valer un 10 a un 10.0, puesto que es lo mismo.

¿Por qué C permite esta operación sin generar un error?

Respuesta: Porque al final **ambos datos son tipos numéricos**, esto también podría pasar con otro tipo de variables que pertenezcan al tipo de dato numérico como podría ser un double o un long. Y debido a que como es normal en la matemática, aunque nosotros estemos viendo un 10 sin más, realmente este contiene un 10.0.

Código 2: Uso de punteros y tipos

```c
#include <stdio.h>
int main() {
    int numero = 42;
    char *puntero = (char *)&numero; // Conversión explícita de int* a char*
    printf("Valor apuntado por 'puntero': %d\n", *puntero);
    return 0;
}
```

Preguntas para reflexión:

¿Qué hace la línea char *puntero = (char *)Numero?

Respuesta: Primero que nada, este tronara por que marca un error, y esto es debido a que es necesario utilizar el **“&”** para que apunte correctamente a la variable. Ahora la intención del problema es darse cuenta de que al utilizar el puntero lo que provocamos es que ya no se pregunte si el tipo de dato es convertible o no, porque simplemente apuntamos al valor del resultado pero no al tipo de dato del resultado, provocando que tomemos directamente su valor y de nuevo, sin tener que hacerlo, convertirlo de manera implícita a otro tipo de dato.

Dato Extra: Si bien no forma parte de la respuesta se me hizo curioso el mencionarlo, y es que si tratamos de asignarle el valor a la variable “numero” de 128 este nos retornara en el printf el número -128. Supongo que esto es debido a que como C esta con arquitectura de bajo nivel procesa los datos en binario y toma el octavo numero binario, como bandera de positivo y negativo provocando que si tenemos un 127 sería un “01111111” el 0 siendo la bandera de los positivos y cuando pasamos a 128 “10000000” el 1 convierte el numero a negativo y volveremos a positivo en el 256 y así sucesivamente. **Esto podría generar un error horrible y que muchos puede que no encontrarían**.

¿Por qué C permite convertir un puntero de un tipo a otro sin restricciones?

Respuesta: Realmente no entendí muy bien por qué pasaba tal cual, hasta que si tuve que ver varios videos y generar una opinión propia al respecto. Y como en mi respuesta anterior es tal cual lo que sucedió. El lenguaje no impone una restricción real para impedir que estas cosas pasen y supone que ya tu sabes que onda. Y

como antes lo mencionaba, realmente no se apunta a un tipo de dato y su valor, únicamente un apuntador toma su valor por lo cual ignora su tipo de dato y únicamente toma lo que necesita, lo cual podría ser peligroso en muchos casos.

Código 3: Ausencia de verificación de tipos

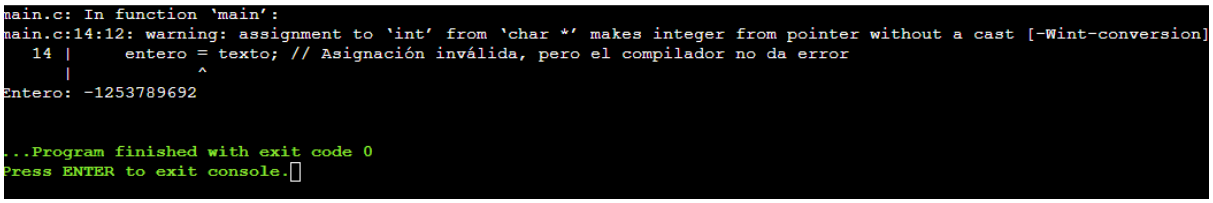
```c
#include <stdio.h>
int main() {
    int entero = 10;
    char *texto = "Hola";

    entero = texto; // Asignación inválida, pero el compilador no da error
    printf("Entero: %d\n", entero);
    return 0;
}
```

Preguntas para reflexión:

¿Por qué el compilador de C no genera un error en la línea entero = texto?

Respuesta: Si bien es cierto que no arroja un error como tal, si me arrojo como un “warning” y es el siguiente:



```
main.c: In function 'main':
main.c:14:12: warning: assignment to 'int' from 'char *' makes integer from pointer without a cast [-Wint-conversion]
   14 |     entero = texto; // Asignación inválida, pero el compilador no da error
      |            ^
Entero: -1253789692


...Program finished with exit code 0
Press ENTER to exit console.
```

Que menciona que la asignación es invalida, pero aun así imprime un numero entero, aunque cada que ejecuto el programa es un numero aleatorio.

Investigando porque como en la anterior pregunta me quede con dudas de si estaba bien. Decidí probar varios compiladores y según logre ver que según compilador que se utilice puede marcar un error o simplemente quedarse en una advertencia.

Como sea, simplemente esto sucede por lo explicado en el anterior punto, realmente C no verifica los tipos de datos cuando se trata de usar apuntadores y únicamente toma su valor como dato, por lo que permite realizar esta acción, pero como logre observar arroja basura.

¿Qué riesgos puede tener esta falta de verificación de tipos?

Respuesta: Simple, somos humanos y en algún momento puede que lleguemos a apuntar una variable que se encuentra declarada en otro tipo de dato como es este caso y lo que provoque o genere de respuesta sea necesario para algo más, tipo si guardamos la cantidad de dinero en una variable de tipo cadena y luego decidimos hacer eso, provoca que siempre aparezca un numero aleatorio y eso seria muy malo o que realmente no debamos manipular un dato como se debería. Y como no aparece ningún error eso podría retrasar muchísimo un proyecto y realmente seria casi imposible en algunos casos encontrar el error o seria necesarios muchos conocimientos para entender este tipo de errores.

Conclusión:

Después de analizar estos ejemplos, divertirme dándome de topes con los errores y reflexionar sobre ellos, me doy cuenta de lo flexible, pero también peligrosa, que puede ser la gestión de tipos en C. Al principio, me costó entender por qué el lenguaje permitía conversiones implícitas sin generar errores, pero ahora veo que todo se debe a su naturaleza de bajo nivel y a la confianza que deposita en el programador (que considero bastante peligrosa, pero no dudo que en algún momento alguien se haya aprovechado de esto).

El hecho de que C no imponga restricciones estrictas en la conversión de tipos y en el uso de punteros hace que sea increíblemente poderoso, pero también riesgoso. Puedo ver cómo errores como la asignación de un puntero a una variable entera o la falta de verificación de tipos pueden generar valores inesperados, corrupción de datos e incluso fallos graves en un programa. Lo más preocupante es que estos errores pueden ser difíciles de detectar, ya que muchas veces el compilador solo emite una advertencia en lugar de un error o incluso según el tipo de compilador que tengas, puede llegar solamente ejecutar el programa como si nada pasara.

Uno de los aspectos que más me llamó la atención es cómo C maneja los números en binario y cómo el cambio de signo en un entero puede ocurrir debido a la forma en que se representan los datos en memoria. Esto me hizo darme cuenta de que, en C, no solo importa qué escribo en el código, sino también cómo se interpretan esos datos en el hardware. Cosas que gracias a lo que ya he aprendido a este punto de la carrera logro entender por que suceden y puede que si me hubieran tratado de explicar esto en el primer semestre de la carrera seria todo un caos.

Pero, en conclusión, este ejercicio me dejó claro que trabajar con C requiere un alto nivel de precisión y comprensión sobre cómo funcionan los tipos de datos y los punteros. Ahora entiendo que la flexibilidad del lenguaje puede ser un arma de doble filo: por un lado, me permite hacer prácticamente lo que quiera con la memoria y los datos, pero por otro, esa misma libertad puede llevarme a errores difíciles de rastrear. Sin duda, esta actividad me ha hecho más consciente de la importancia de entender la naturaleza del lenguaje como también en prestar atención a las advertencias del compilador (que muchas veces se suelen ignorar porque si el programa jala, ya no se le mueve).