Sesión 3: Tipos básicos

Hoja de problemas

Programación para Sistemas

Ángel Herranz
aherranz@fi.upm.es
Universidad Politécnica de Madrid

2022-2023

Ejercicio 1. Repasa las transparencias de clase. Ten siempre en cuenta que en cada variable sólo puede haber `0`s y `1`s.

Ejercicio 2. Asegúrate de realizar todos los ejercicios de las transparencias antes de continuar.

Ejercicio 3. Ya has visto que es complicado fiarse de lo que `printf` nos enseña. Utiliza `gdb` para ver el contenido *exacto* de la variable `mi_double` en el programa `basicos.c`.

Este ejercicio es especialmente complicado, así que aquí tienes una sesión de `gdb` que puede ayudarte:

```
angel@T440p: /03-basicos (master)$ gdb basicos
GNU gdb (Ubuntu 8.1-0ubuntu3.1) 8.1.0.20180409-git
[...]
Reading symbols from basicos...done.
(gdb) break main
Breakpoint 1 at 0x652: file basicos.c, line 3.
(gdb) run
Starting program: /home/angel/Asignaturas/undergit/pps-src/03-basicos/basicos

Breakpoint 1, main () at basicos.c:3
3       char mi_char = 'a';
(gdb) step
4       int mi_int = 42;
(gdb) step
5       float mi_float = 10000000.0;
(gdb) step
6       double mi_double = 0.0000001;
(gdb) step
7       printf("El char es: %c\n", mi_char);
(gdb) x/tg &mi_double
0x7fffffffdfb8: 0011111100111110101011010011111111001010011010111100101000
(gdb) q
angel@T440p: /03-basicos (master)$
```

Ejercicio 4. Los programadores somos muy vagos. No soportamos tener que teclear

```
$ gcc -ansi -Wall -Werror -pedantic -o basicos basicos.c
```

cada vez que necesitamos compilar el programa `basicos.c` para generar el ejecutable `basicos`. Para evitar este tipo de tareas tan tediosas, y evitar también errores a la hora de escribir esas líneas, tenemos herramientas que nos ayudan.

Una de estas herramientas es `make`. Cuando ejecutamos `make`, la herramienta busca un fichero `Makefile` en el que hay una serie de **reglas** con las que automatizamos ciertas tareas. Veamos un ejemplo:

```
1 basicos: basicos.c
2     gcc -ansi -Wall -Werror -pedantic -o basicos basicos.c
```

La primera línea dice: para conseguir el fichero `basicos` necesitas el fichero `basicos.c`. La segunda línea dice: para construir el fichero `basicos` hay que ejecutar el mandato

```
$ gcc -ansi -Wall -Werror -pedantic -o basicos basicos.c
```

A partir de ahí, basta con ejecutar la línea

```
$ make basicos
```

Quizás lo más importante: **`make` *sabe* que sólo es necesario ejecutar el mandato cuando `basicos` es *más viejo* que `basicos.c`**.

Ejercicio 5. Los ficheros `Makefile` admiten que pongas tantas reglas como quieras. Eso significa que para practicar con los programas de cada clase puedes hacerte un `Makefile` con las reglas que *fabriquen* todos los programas. Por ejemplo:

```
1 basicos: basicos.c
2     gcc -ansi -Wall -Werror -pedantic -o basicos basicos.c
3
4 sizeof: sizeof.c
5     gcc -ansi -Wall -Werror -pedantic -o sizeof sizeof.c
```

No dejes de crear ese `Makefile` y de probarlo.

Ejercicio 6. Seguro que has oido hablar del "código ASCII". Ha llegado el momento de entenderlo `;)`. Busca en Internet lo que es el código ASCII, lo necesitas para el siguiente ejercicio.

Ejercicio 7. Escribe un programa (`caracteres.c`) que escriba en la salida estándar los caracteres imprimibles del código ASCII junto con su código en decimal.

```
' ', 32
'!', 33
...
'}', 125
'~', 126
```

Tu programa debería imprimir algo como

Ejercicio 8. Escribe un programa y haz las modificaciones necesarias para conocer los

valores máximo y mínimo de los siguientes tipos (no uses la biblioteca `limits.h`): **`char`**, **`unsigned char`**, **`int`**, **`unsigned int`**, **`long int`**, **`unsigned long int`**, **`long long int`**, **`unsigned long long int`**.

Ejercicio 9. A la luz de lo que has visto en el ejercicio anterior, deduce el tamaño en bytes de los tipos (no uses el operador **`sizeof`**).

Ejercicio 10. Contrasta que tus respuestas a los ejercicios anteriores son correctas: escribe un programa que imprima en la salida estándar el nombre de cada tipo, el tamaño de sus valores en bytes y los valores máximo y mínimo.

```
...
unsigned 1 0 255
char 1 -128 127
Tu programa debería imprimir algo como
```

Ejercicio 11. Asegúrate que entiendes todas tus deducciones sobre límites y tamaños de los tipos.

Ejercicio 12. Ejecuta `man 3` **`printf`** y muévete por el manual de *printf* entendiendo su estructura.

Ejercicio 13. Lee las secciones *Conversion specifiers* y *Length modifier* del manual anterior. Presta especial atención a los formatos

`%i, %u, %d, %o, %x, %e, %E, %f, %F, %s, %c`

y a los modificadores

`%l, %h.`

```
Conversion specifiers
    A character that specifies the type of conversion to be
    applied.  The conversion specifiers and their meanings are:

    d, i   The int argument is converted to signed decimal nota-
           tion.  The precision, if any, gives the minimum number
           of digits that must appear; if the converted value
           requires fewer digits, it is padded on the left with
           zeros.  The default precision is 1.  When 0 is printed
           with an explicit precision 0, the output is empty.

    o, u, x, X
           The unsigned int argument is converted to unsigned octal
           (o), unsigned decimal (u), or unsigned hexadecimal (x
           and X) notation.  The letters abcdef are used for x con-
           versions; the letters ABCDEF are used for X conversions.
           The precision, if any, gives the minimum number of dig-
           its that must appear; if the converted value requires
           fewer digits, it is padded on the left with zeros.  The
           default precision is 1.  When 0 is printed with an
           explicit precision 0, the output is empty.
```

e, E The double argument is rounded and converted in the style [-]d.ddde±dd where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An E conversion uses the letter E (rather than e) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.

f, F The double argument is rounded and converted to decimal notation in the style [-]ddd.ddd, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

(SUSv2 does not know about F and says that character string representations for infinity and NaN may be made available. SUSv3 adds a specification for F. The C99 standard specifies "[-]inf" or "[-]infinity" for infinity, and a string starting with "nan" for NaN, in the case of f conversion, and "[-]INF" or "[-]INFINITY" or "NAN" in the case of F conversion.)

g, G The double argument is converted in style f or e (or F or E for G conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style e is used if the exponent from its conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

a, A (C99; not in SUSv2, but added in SUSv3) For a conversion, the double argument is converted to hexadecimal notation (using the letters abcdef) in the style [-]0xh.hhhhp±; for A conversion the prefix 0X, the letters ABCDEF, and the exponent separator P is used. There is one hexadecimal digit before the decimal point, and the number of digits after it is equal to the precision. The default precision suffices for an exact representation of the value if an exact representation in base 2 exists and otherwise is sufficiently large to distinguish values of type double. The digit before the decimal point is unspecified for nonnormalized numbers, and nonzero but otherwise unspecified for normalized numbers.

c If no l modifier is present, the int argument is converted to an unsigned char, and the resulting character is written. If an l modifier is present, the wint_t (wide character) argument is converted to a multibyte sequence by a call to the wcrtomb(3) function, with a conversion state starting in the initial state, and the resulting multibyte string is written.

s If no l modifier is present: the const char * argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating null byte ('\0'); if a precision is specified, no more than the number specified are written. If a precision is given, no null byte need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating null byte.

If an l modifier is present: the const wchar_t * argument is expected to be a pointer to an array of wide characters. Wide characters from the array are converted to multibyte characters (each by a call to the wcrtomb(3) function, with a conversion state starting in the initial state before the first wide character), up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null byte. If a precision is specified, no more bytes than the number specified are written, but no partial multibyte characters are written. Note that the precision determines the number of bytes written, not the number of wide characters or screen positions. The array must contain a terminating null wide character, unless a precision is given and it is so small that the number of bytes written exceeds it before the end of the array is reached.

C (Not in C99 or C11, but in SUSv2, SUSv3, and SUSv4.) Synonym for lc. Don't use.

S (Not in C99 or C11, but in SUSv2, SUSv3, and SUSv4.) Synonym for ls. Don't use.

p The void * pointer argument is printed in hexadecimal (as if by %#x or %#lx).

n The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an int *, or variant whose size matches the (optionally) supplied integer length modifier. No argument is converted. (This specifier is not supported by the bionic C library.) The behavior is undefined if the conversion specification includes any flags, a field width, or a precision.

m (Glibc extension; supported by uClibc and musl.) Print output of strerror(errno). No argument is required.

% A '%' is written. No argument is converted. The complete conversion specification is '%%'.

Length modifier

Here, "integer conversion" stands for d, i, o, u, x, or X conversion.

hh A following integer conversion corresponds to a signed char or unsigned char argument, or a following n conver-

```
       sion corresponds to a pointer to a signed char argument.

   h   A following integer conversion corresponds to a short
       int or unsigned short int argument, or a following n
       conversion corresponds to a pointer to a short int argu-
       ment.

   l   (ell) A following integer conversion corresponds to a
       long int or unsigned long int argument, or a following n
       conversion corresponds to a pointer to a long int argu-
       ment, or a following c conversion corresponds to a
       wint_t argument, or a following s conversion corresponds
       to a pointer to wchar_t argument.

   ll  (ell-ell). A following integer conversion corresponds
       to a long long int or unsigned long long int argument,
       or a following n conversion corresponds to a pointer to
       a long long int argument.

   q   A synonym for ll. This is a nonstandard extension,
       derived from BSD; avoid its use in new code.

   L   A following a, A, e, E, f, F, g, or G conversion corre-
       sponds to a long double argument. (C99 allows %LF, but
       SUSv2 does not.)

   j   A following integer conversion corresponds to an int-
       max_t or uintmax_t argument, or a following n conversion
       corresponds to a pointer to an intmax_t argument.

   z   A following integer conversion corresponds to a size_t
       or ssize_t argument, or a following n conversion corre-
       sponds to a pointer to a size_t argument.

   Z   A nonstandard synonym for z that predates the appearance
       of z. Do not use in new code.

   t   A following integer conversion corresponds to a
       ptrdiff_t argument, or a following n conversion corre-
       sponds to a pointer to a ptrdiff_t argument.
```

Ejercicio 14. Una vez conoces los tamaños de los tipos y los límites, puedes forzar *overflows*. Escribe código para ver qué pasa cuando intentas ir más allá del límite (por ejemplo sumando o restando 1 a un límite.

Ejercicio 15. No dejes de leer el artículo de David Goldberg: *What every computer scientist should know about floating-point arithmetic*

Ejercicio 16. Sorprendéntemente puedes pedirle a C que te imprima un dato como si fuera de otro tipo. Por ejemplo, imprimir un caracter como si fuera un entero o un entero negativo como si fuera **`unsigned`** o incluso un número en coma flotante como si fuera un entero. El compilador se va a quejar y no podrás compilar alguno de esos intentos, especialmente si usas los *flags* `-Werror`, no lo uses para compilar este código:

```
int i = -42;
```

```
  float f = -3.0;
  printf("%d\n", i);
  printf("%u\n", i);
  printf("%f\n", i);
  printf("%d\n", f);
  printf("%u\n", f);
  printf("%f\n", f);
```

El compilador debería decir algo como:

```
In function ‘main’:
warning: format ‘%f’ expects argument of type ‘double’,
   but argument 2 has type ‘int’ [-Wformat=]
   printf("%f\n", i);
           ~^
           %d
warning: format ‘%d’ expects argument of type ‘int’,
   but argument 2 has type ‘double’ [-Wformat=]
   printf("%d\n", f);
           ~^
           %f
warning: format ‘%u’ expects argument of type ‘unsigned int’,
   but argument 2 has type ‘double’ [-Wformat=]
   printf("%u\n", f);
           ~^
           %f
```

Y la salida del programa debería ser:

```
-42
4294967254
0.000000
-124652960
4170314336
-3.000000
```

💬 **Ejercicio 17.** ¿Puedes darle una interpretación a esos datos?

⏭ **Ejercicio 18.**

No nos gustan nada los *warnings*

Son un indicador de que algo no va mal o de que no somos conscientes de algo. Por eso usamos el *flag* `-Werror`. Vamos a adelantarnos un poco y hacer conversiones de tipos que eliminen las quejas del compilador. Por ejemplo, si C espera un **`float`** y le pasamos un entero `n`, podemos pedir al compilador que *convierta* (*cast*) `n` a **`float`** con esta expresión: `(`**`float`**`)n`. Prueba a eliminar los *warnings*.

💻 **Ejercicio 19.** Ejecuta ahora el programa y compara los resultados con los del programa sin

las conversiones.

⏭ **Ejercicio 20.** Transcribe el siguiente programa e interpreta su salida:

```
#include <stdio.h>
int main() {
  int x = 42;
  char *s = "Un string en C";
  printf("%p\n",(void *)&x);
  printf("%p\n",(void *)&s);
  printf("%p\n",s);
  return 0;
}
```