El Lenguaje C

Eduardo Grosclaude

2014-12-09

[2014/12/10, 11:03:41- Material en preparación, se ruega no imprimir mientras aparezca esta nota]

Índice general

Capítulo 1

Introducción al Lenguaje C

El lenguaje de programación C fue creado por **Dennis Ritchie** en 1972 en Bell Telephone Laboratories, con el objetivo de reescribir un sistema operativo, el UNIX, en un lenguaje de alto nivel, para poder adaptarlo (es decir, *portarlo*) a diferentes arquitecturas. Por este motivo sus creadores se propusieron metas de diseño especiales, tales como:

- Poder utilizar todos los recursos del hardware ("acceso al bajo nivel").
- Obtener código generado eficiente en uso de memoria y en tiempo de ejecución (programas pequeños y veloces).
- Compilador portable (implementable en cualquier arquitectura).

Actualmente existen implementaciones de C para todas las arquitecturas y sistemas operativos, y es el lenguaje más utilizado para la **programación de sistemas**. Por su gran eficiencia resulta ideal para la programación de **sistemas operativos**, **drivers de dispositivos**, **herramientas de programación**. El 95 % del sistema operativo UNIX está escrito en C, así como gran parte de los modernos sistemas y ambientes operativos, y los programas de administración o aplicación que corren sobre ellos.

1.1. Características del lenguaje

C es un lenguaje compilado. A nivel sintáctico, presenta grandes similitudes formales con Pascal, pero las diferencias entre ambos son importantes. A pesar de permitir **operaciones de bajo nivel**, tiene las **estructuras de control**, y permite la **estructuración de datos**, propias de los lenguajes procedurales de alto nivel.

Un programa en C es, por lo general, más **sintético** que en otros lenguajes procedurales; la idea central que atraviesa todo el lenguaje es la minimalidad. La definición del lenguaje consta de muy pocos elementos, y tiene muy pocas **palabras reservadas**. Como rasgo distintivo, en C no existen, rigurosamente hablando, funciones o procedimientos de uso general del programador. Por ejemplo, **no tiene funciones de entrada/salida**; la definición del lenguaje apenas alcanza a **las estructuras de control y los operadores**. La idea de definir un lenguaje sin funciones es, por un lado, hacer posible que el compilador sea **pequeño, fácil de escribir e inmediatamente portable**; y por otro, permitir que sea el usuario quien defina sus propias funciones cuando el problema de programación a resolver tenga requerimientos especiales.

Sin embargo, se ha establecido un conjunto mínimo de funciones, llamado la **Biblioteca Standard** del lenguaje C, que todos los compiladores proveen, a veces con agregados. La filosofía de la Biblioteca Standard es la portabilidad, es decir, casi no incluye funciones que sean específicas de un sistema operativo determinado. Aquellas que sí incluye están orientadas a la programación de sistemas, y a veces no resultan

suficientes para el programador de aplicaciones. No provee, por ejemplo, la capacidad de manejo de archivos indexados, ni funciones de entrada/salida interactiva por consola que sean seguras ("a prueba de usuarios"). Esta deficiencia se remedia utilizando bibliotecas de funciones "de terceras partes" (creadas por el usuario u obtenidas de otros programadores).

El usuario puede escribir sus propios procedimientos (llamados **funciones** aunque no devuelvan valores). Aunque existe la noción de **bloque** de sentencias (sentencias encerradas entre llaves), el lenguaje se dice ***no* estructurado en bloques** porque no pueden definirse funciones dentro de otras. Las funciones de la Biblioteca Standard no tienen ningún privilegio sobre las del usuario y **sus nombres no son palabras reservadas**; el usuario puede reemplazarlas por sus propias funciones simplemente dándoles el mismo nombre.

El lenguaje entrega completamente el control de la máquina subyacente al programador, no realizando controles de tiempo de ejecución. Es decir, no verifica condiciones de error comunes como ***overflow* de variables**, **errores de entrada/salida**, o **consistencia de argumentos** en llamadas a funciones. Como resultado, es frecuente que el principiante, y aun el experto, cometan errores de programación que no se hacen evidentes enseguida, ocasionando problemas y costos de desarrollo. Permite una gran libertad sintáctica al programador. No es fuertemente tipado. Cuando es necesario, se realizan **conversiones automáticas de tipo** en las asignaciones, a veces **con efectos laterales inconvenientes** si no se tiene precaución. Una función que recibe determinados parámetros formales puede ser invocada con argumentos reales de otro tipo.

Se ha dicho que estas características "liberales" posibilitan la realización de proyectos complejos con más facilidad que otros lenguajes como Pascal o Ada, más estrictos; aunque al mismo tiempo, así resulta más difícil detectar errores de programación en tiempo de compilación. En este sentido, según los partidarios de la tipificación estricta, C no es un buen lenguaje. Gran parte del esfuerzo de desarrollo del estándar ANSI se dedicó a dotar al C de elementos para mejorar esta deficiencia.

Los **tipos de datos** no tienen un tamaño determinado por la definición del lenguaje, sino que diferentes implementaciones pueden adoptar diferentes convenciones. Paradójicamente, esta característica obedece al objetivo de lograr la **portabilidad** de los programas en C. El programador está obligado a no hacer ninguna suposición sobre los tamaños de los objetos de datos, ya que lo contrario haría al software dependiente de una arquitectura determinada ("no portable").

Una característica especial del lenguaje C es que el **pasaje de argumentos a funciones** se realiza siempre **por valor**. ¿Qué ocurre cuando una función debe **modificar** datos que recibe como argumentos? La única salida es pasarle -por valor- la dirección del dato a modificar. Las consecuencias de este hecho son más fuertes de lo que parece a primera vista, ya que surge la necesidad de todo un conjunto de técnicas de **manejo de punteros** que no siempre son bien comprendidas por los programadores poco experimentados, y abre la puerta a sutiles y escurridizos errores de programación. Quizás este punto, junto con el de la ausencia de chequeos en tiempo de ejecución, sean los que le dan al C fama de "difícil de aprender".

Por último, el C **no es un lenguaje orientado a objetos**, sino que adhiere al paradigma tradicional de **programación procedural**. No soporta la orientación a objetos propiamente dicha, al no proporcionar herramientas fundamentales, como la herencia. Sin embargo, algunas características del lenguaje permiten que un proyecto de programación se beneficie de todas maneras con la aplicación de algunos principios de la orientación a objetos, tales como el ocultamiento de información y el encapsulamiento de responsabilidades. El lenguaje C++, orientado a objetos, **no** es una versión más avanzada del lenguaje o un compilador de C con más capacidades, sino que **se trata de un lenguaje completamente diferente**.

1.2. Evolución del lenguaje

La primera definición oficial del lenguaje fue dada en 1978 por **Brian Kernighan y Dennis Ritchie** (Fig. 1.1) en su libro **El lenguaje de programación C**. Este lenguaje fue llamado **C K&R**, por las iniciales

de sus autores. En 1983 se creó el comité ANSI para el lenguaje, que en 1988 estableció el estándar ANSI C, con algunas reformas sobre el C K&R. Simultáneamente, Kernighan y Ritchie publicaron la segunda edición de su libro, describiendo la mayor parte de las características del ANSI C.

Figura 1.1: Brian Kernighan y Dennis Ritchie, los creadores del lenguaje C

Algunas nuevas características de C99 son:

- Matrices de tamaño variable
- Soporte de números complejos
- Tipos `long long int` y `unsigned long long int` de al menos 64 bits
- Familia de funciones `vscanf()`
- Comentarios al estilo de C++ prefijando las líneas con la secuencia `//`.
- Familia de funciones `snprintf()`
- Tipo boolean

1.3. El ciclo de compilación

Las herramientas esenciales de un ambiente de desarrollo, además de cualquier **editor de textos**, son el **compilador**, el **vinculador**, **linkeditor** o *linker*, y el **bibliotecario**. A estas herramientas básicas se agregan otras, útiles para automatizar la compilación de proyectos extensos, almacenar y recuperar versiones de programas fuente, comprobar sintaxis en forma previa a la compilación, etc. Según el ambiente operativo y producto de software de que se trate, estas herramientas pueden ser comandos de línea independientes, con salidas de texto simples, o encontrarse integradas en una interfaz de usuario uniforme, en modo texto o modo gráfico.

Cuando encontramos varias de estas herramientas integradas en una sola aplicación, decimos que se trata de un **IDE** (*Integrated Development Environment*) o ambiente de desarrollo integrado. Un IDE oculta el ciclo de compilación al usuario, con la intención de simplificar el proceso de desarrollo. Sin embargo, conviene conocer qué función se cumple, y qué producto se espera, en cada fase del ciclo de compilación, para poder interpretar las diferentes situaciones de error y poder corregirlas.

Compilador

- El compilador acepta un archivo **fuente**, posiblemente relacionado con otros (una **unidad de traducción**), y genera con él un **módulo objeto**. Este módulo objeto contiene porciones de código

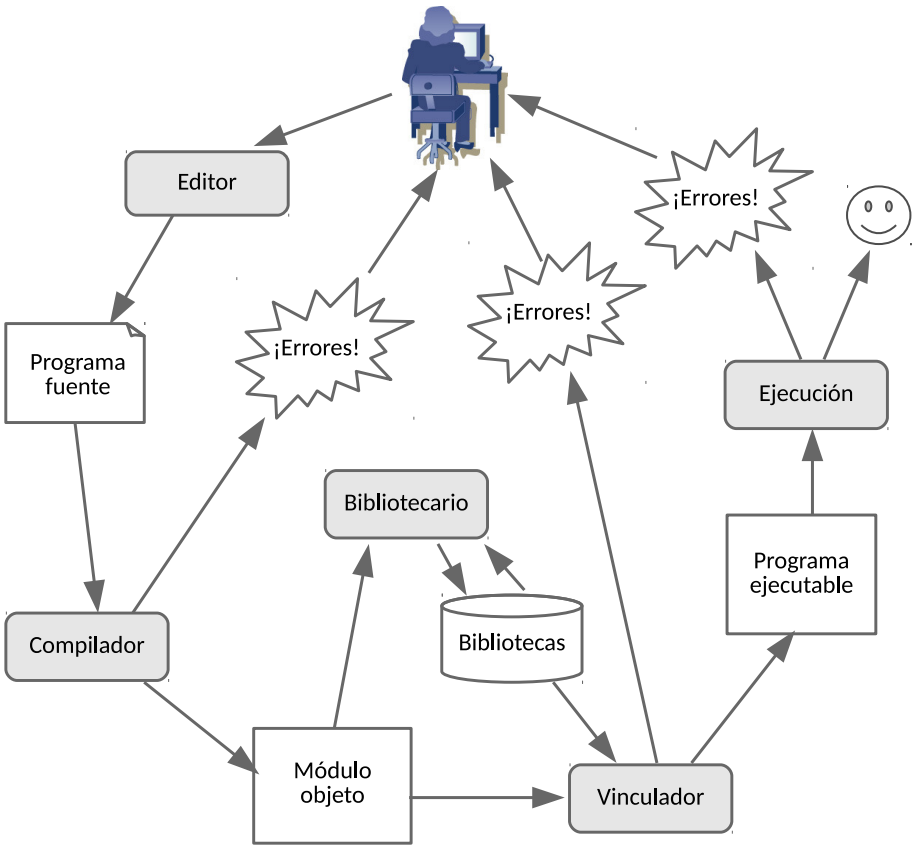

Figura 1.2: El ciclo de compilación produce un ejecutable a partir de archivos fuente.

ejecutable mezclado con **referencias**, aún no resueltas, a variables o funciones cuya definición no figura en los fuentes de entrada. Estas referencias quedan en forma simbólica en el módulo objeto hasta que se resuelvan en un paso posterior.

- Si ocurren errores en esta fase, se deberán a problemas de sintaxis (el código escrito por el programador no respeta la definición del lenguaje).

Vinculador, linkeditor o *linker*

- El vinculador recibe como entrada un conjunto de módulos objeto y busca **resolver**, vincular, o enlazar, las referencias simbólicas en ellos, buscando la definición de las variables o funciones faltantes en los mismos objetos o en bibliotecas. Éstas pueden ser la Biblioteca Standard, u otras provistas por el usuario. Cuando el linker encuentra la definición de un objeto buscado (es decir, de una variable o función), la copia en el archivo resultante de salida (la *resuelve*). El objetivo del linker es resolver todas las referencias pendientes para producir un programa ejecutable.

- Si ocurren errores en esta fase, se deberán a que existen variables o funciones cuya definición no ha sido dada (no se encuentran en las unidades de traducción procesadas, ni en ninguna biblioteca conocida por el linker).

Bibliotecario

- El bibliotecario es un administrador de módulos objeto. Su función es reunir módulos objeto en archivos llamados **bibliotecas**, y luego permitir la extracción, borrado, reemplazo y agregado de módulos. El conjunto de módulos en una biblioteca se completa con una tabla de información sobre sus contenidos para que el linker pueda encontrar rápidamente aquellos módulos donde se ha definido una variable o función, y así extraerlos durante el proceso de linkedición.
- El bibliotecario es utilizado por el usuario cuando desea mantener sus propias bibliotecas. La creación de bibliotecas propias del usuario ahorra tiempo de compilación y permite la distribución de software sin revelar la forma en que se han escrito los fuentes y protegiéndolo de modificaciones.

Una vez que el código ha sido compilado y vinculado, obtenemos un programa ejecutable. Los errores que pueden producirse en la ejecución ya no corresponden a problemas de compilación, sino que se deben a aspectos de diseño del programa que deben ser corregidos por el programador.

1.4. Un primer ejemplo

El clásico ejemplo de todas las introducciones al lenguaje C es un programa llamado "**Hello, World!**".

```
#include <stdio.h>
/* El primer programa! */
main()
{
        printf("Hola, gente!\n");
}
```

Funcionamiento del programa

- Este programa minimal comienza con una **directiva de preprocesador** que indica incluir en la unidad de traducción al archivo de cabecera o *header* **stdio.h**. Éste contiene, entre otras cosas, la declaración (o **prototipo**) de la función de salida de caracteres **printf()**, perteneciente a la Biblioteca Standard. Los prototipos se incluyen para advertir al compilador de los tipos de las funciones y de sus argumentos.
- Entre los pares de caracteres especiales **/*** y ***/**se puede insertar cualquier cantidad de líneas de comentarios.
- La función **main()** es el cuerpo principal del programa (es por donde comenzará la ejecución). Todas las funciones en C están delimitadas por un par de llaves. Terminada la ejecución de main(), terminará el programa.
- La función `printf()` imprimirá la cadena entre comillas, que es una **constante string** terminada por un carácter de **nueva línea** (la secuencia especial "`\n`").

Compilación del programa

Para ver el primer ejemplo en C en funcionamiento:

1. Copiar el programa con cualquier editor de textos y guardarlo en un archivo llamado `hola.c` en el directorio de trabajo del usuario.

2. Sin cambiar de directorio, ejecutar el comando `make hola` en una consola o terminal.
3. Ejecutar el programa con el comando `./hola`. Notar el *punto y barra* del principio, que le indican al **shell** que debe buscar el programa en el directorio activo.

Lo mismo, pero de otra manera:

1. Como antes, copiar el programa, o usar el mismo archivo fuente de hace un momento.
2. Sin cambiar de directorio, ejecutar el comando `gcc hola.c -o hola`.
3. Ejecutar el programa con el comando `./hola`.

La diferencia es que en el primer caso utilizamos la herramienta `make`, que nos asiste en la compilación de proyectos, mientras que en el segundo caso invocamos directamente al compilador `gcc`. Por defecto, el compilador gcc invocará al vinculador para generar el ejecutable a partir del archivo objeto intermedio generado. En el ejemplo, le decimos al compilador que procese el archivo fuente `hola.c`, y que el ejecutable de salida (opción `-o`, de *output*) se llame `hola`.

1.5. Ejercicios

1. ¿Qué nombres son adecuados para los archivos fuente C?
2. Describa las etapas del ciclo de compilación.
3. ¿Cuál sería el resultado de:
   - Editar un archivo fuente?
   - Ejecutar un archivo fuente?
   - Editar un archivo objeto?
   - Compilar un archivo objeto?
   - Editar una biblioteca?
4. ¿Qué pasaría si un programa en C **no** contuviera una función **main()**? Haga la prueba modificando **hola.c**.
5. Edite el programa `hola.c` y modifíquelo según las pautas que siguen. Interprete los errores de compilación. Identifique en qué etapa del ciclo de compilación ocurren los errores. Si resulta un programa ejecutable, observe qué hace el programa y por qué.
   - Quite los paréntesis de main().
   - Quite la llave izquierda de main().
   - Quite las comillas izquierdas.
   - Quite los caracteres "\n".
   - Agregue al final de la cadena los caracteres "\n\n\n\n".
   - Agregue al final de la cadena los caracteres "\nAdiós, mundo!\n".
   - Quite las comillas derechas.
   - Quite el signo punto y coma.
   - Quite la llave derecha de main().

- Agregue un punto y coma en cualquier lugar del texto.
- Agregue una coma o un dígito en cualquier lugar del texto.
- Reemplace la palabra **main** por **program**, manteniendo los paréntesis.
- Elimine la apertura o cierre de los comentarios.

Ejercicios Adicionales

Ejercicios Avanzados

Capítulo 2

El preprocesador

El compilador C tiene un componente auxiliar llamado **preprocesador**, que actúa en la primera etapa del proceso de compilación. Su misión es buscar, en el texto del programa fuente entregado al compilador, ciertas **directivas** que le indican realizar alguna tarea a nivel de texto. Por ejemplo, **inclusión** de otros archivos, o **sustitución** de ciertas cadenas de caracteres (**símbolos** o **macros**) por otras.

El preprocesador cumple estas directivas en forma similar a como podrían ser hechas interactivamente por el usuario, utilizando los comandos de un editor de texto ("incluir archivo" o "reemplazar texto"), pero en forma automática. Una vez cumplidas todas estas directivas, el preprocesador entrega el texto resultante al resto de las etapas de compilación, que terminarán dando por resultado un módulo objeto.

Directivas de preprocesador

El preprocesador sirve para eliminar redundancia y aumentar la expresividad de los programas en C, facilitando su mantenimiento. Si una variable o función se utiliza en varios archivos fuente, es posible aislar su declaración, colocándola en un único archivo aparte que será incluido al tiempo de compilación en los demás fuentes. Esto facilita toda modificación de elementos comunes en los fuentes de un proyecto. Por otro lado, si una misma constante o expresión aparece repetidas veces en un texto, y es posible que su valor deba cambiarse más adelante, es muy conveniente definir esa constante con un símbolo y especificar su valor sólo una vez, mediante un símbolo o macro.

Una de las funciones del preprocesador es sustituir símbolos, o cadenas de texto dadas, por otras (Fig. 2.1). La directiva `#define` establece la relación entre los símbolos y su expansión o cadena a sustituir. Los símbolos indicados con una directiva de definición `#define` se guardan en una **tabla de símbolos** durante el preprocesamiento. Habitualmente se llama **símbolos** a aquellas cadenas que son directamente sustituibles por una expresión, reservándose el nombre de **macros** para aquellos símbolos cuya expansión es parametrizable (es decir, llevan argumentos formales y reales como en el caso de las funciones). La cadena de expansión puede ser cualquiera, no necesariamente un elemento sintácticamente válido de C.

Las directivas del preprocesador no pertenecen al lenguaje C en un sentido estricto. El preprocesador **no comprende ningún aspecto sintáctico ni semántico** de C. Las **macros** definidas en un programa C **no son variables ni funciones**, sino simplemente cadenas de texto que serán sustituidas por otras. Las directivas pueden aparecer en cualquier lugar del programa, pero sus efectos se ponen en vigor recién a partir del punto del programa en que aparecen y hasta el final de la unidad de traducción. Es decir, un símbolo o macro puede utilizarse después de la aparición de la directiva que la define, y no antes. Tampoco puede utilizarse en una unidad de traducción diferente (los símbolos de preprocesador no se "propagan.^{en}tre unidades de traducción salvo por el uso de directivas de inclusión).

Las directivas para incluir archivos suelen darse al principio de los programas, porque en general se desea que su efecto alcance a todo el archivo fuente. Por esta razón los archivos preparados para ser incluidos

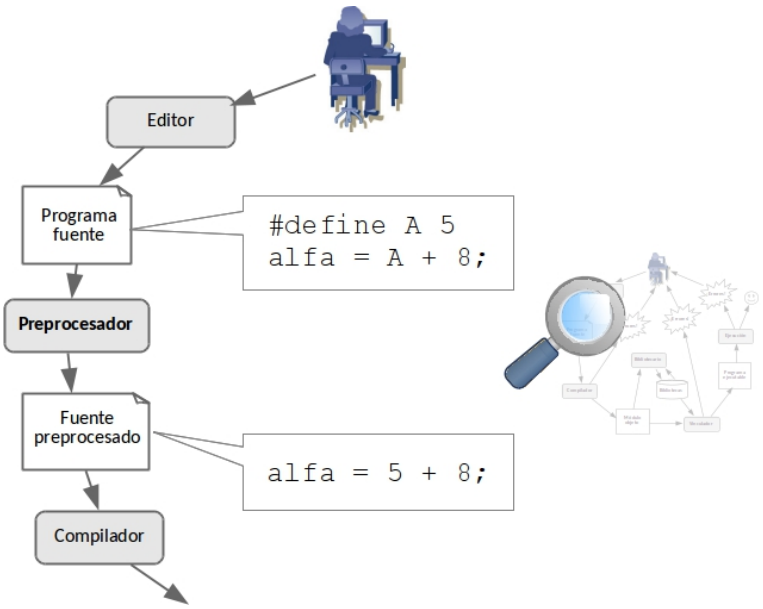

Figura 2.1: El preprocesador realiza ediciones automáticas de los fuentes antes de entregar el resultado al compilador.

se denominan headers o archivos de cabecera. La implementación de la Biblioteca Standard que viene con un compilador posee sus propios headers, uno por cada módulo de la biblioteca, que declaran funciones y variables de uso general. Estos headers contienen texto legible por humanos, y están en algún subdirectorio predeterminado (llamado /usr/include en UNIX, y dependiendo del compilador en otros sistemas operativos). El usuario puede escribir sus propios headers, y no necesita ubicarlos en el directorio reservado del compilador; puede almacenarlos en el directorio activo durante la compilación. Un archivo fuente, junto con todos los archivos que incluya, es llamado una unidad de traducción.

En los párrafos anteriores, nótese que decimos declarar funciones, y no definirlas; la diferencia es importante y se verá con detalle más adelante. Recordemos por el momento que **en los headers** de la Biblioteca Standard no aparecen **definiciones** -es decir, textos- de funciones, sino solamente **declaraciones o prototipos**, que sirven para anunciar al compilador los tipos y cantidad de los argumentos, etc.

Ejemplos

No se considera buena práctica de programación colocar la definición de una función de uso frecuente en un header. Esto forzaría a recompilar siempre la función cada vez que se la utilizara. Por el contrario, lo ideal sería compilarla una única vez, produciendo un módulo objeto (y posiblemente incorporándolo a una biblioteca). Esto ahorraría el tiempo correspondiente a su compilación, ocupando sólo el necesario para la linkedición.

La directiva `#include` hace que el preprocesador inserte y preprocese otros archivos en el punto donde se indica la directiva. El resultado de preprocesar el archivo incluido puede ser definir otros símbolos y macros, o aun incluir otros archivos. Los archivos destinados a ser incluidos son habitualmente llamados archivos de cabecera o headers. Las directivas de inclusión son anidables, es decir, pueden incluirse headers que a su vez contengan directivas de inclusión.

Una característica interesante del preprocesador es que permite la **compilación condicional** de segmentos de la unidad de traducción, en base a valores de símbolos. Una directiva condicional es aquella que comprueba si un símbolo dado ha sido definido, o si su definición coincide con cierta cadena. El texto del

programa que figura entre la directiva y su `end` será considerado sólo si la comprobación resulta exitosa. Los símbolos o macros pueden ser definidos al tiempo de la compilación, sin alterar el texto del programa, permitiendo así una parametrización del programa en forma separada de su escritura.

Ejemplos

1. Si el programa dice:

```
a=2*3.14159*20.299322;
```

   Es mucho más claro poner:

```
#define PI
#define RADIO
a=2*PI*RADIO;
3.14159
20.299322
```

2. Con las siguientes directivas:

```
#include <stdio.h>
#include "aux.h"
#define MAXITEM
#define DOBLE(X)
100
2*X
```

   - Se incluye el header de biblioteca standard `stdio.h`, que contiene declaraciones necesarias para poder utilizar funciones de entrada/salida standard (hacia consola y hacia archivos).
   - Se incluye un header escrito por el usuario. Al indicar el nombre del header entre ángulos, como en la línea anterior, especificamos que la búsqueda debe hacerse en los directorios reservados del compilador. Al indicarlo entre comillas, nos referimos al directorio actual.
   - Se define un símbolo MAXITEM equivalente a la constante numérica 100.
   - Se define una macro DOBLE(X) que deberá sustituirse por la cadena 2*(argumento de la llamada a la macro).

   De esta manera, podemos escribir sentencias tales como:

```
a=MAXITEM;
b=DOBLE(45);
```

El texto luego de la etapa de preprocesamiento y antes de la compilación propiamente dicha será

```
a=100;
b=2*45;
```

Es importante comprender que, aunque sintácticamente parecido, el uso de una macro **no es una llamada a función**; los argumentos de una macro no se evalúan en tiempo de ejecución antes de la llamada, sino que **se sustituyen textualmente** en el cuerpo de la macro. Así, si ponemos

```
b=DOBLE(40+5);
```

el resultado será b=2*40+5; y no b=2*45, ni b=2*(40+5), que presumiblemente es lo que desea el programador.

Este problema puede solucionarse redefiniendo la macro así:

```
#define DOBLE(X)
2*(X)
```

Ahora la expansión de la macro será la deseada. En general es saludable rodear las apariciones de los argumentos de las macros entre paréntesis, para obligar a su evaluación al tiempo de ejecución con la precedencia debida, y evitar efectos laterales. 3) Con las directivas condicionales:

```
#ifdef DEBUG
#define CARTEL(x)
#else
#define CARTEL(x)
#endif
```

Observaciones

```
imprimir(x)
#if SISTEMA==MS_DOS
#include "header1.h"
#elif SISTEMA==UNIX
#include "header2.h"
#endif
```

En el primer caso, definimos una macro CARTEL que equivaldrá a invocar a una función 'imprimir', pero sólo si el símbolo DEBUG ha sido definido. En otro caso, equivaldrá a la cadena vacía. En el segundo caso, se incluirá uno u otro header dependiendo del valor del símbolo SISTEMA. Tanto DEBUG como SISTEMA pueden tomar valores al momento de compilación, si se dan como argumentos para el compilador. De esta manera se puede modificar el comportamiento del programa sin necesidad de editarlo. 4) El segmento siguiente muestra un caso con lógica inversa pero equivalente al ejemplo anterior. ifndef DEBUG define CARTEL(x) else define CARTEL(x) imprimir(x) endif Observaciones A veces puede resultar interesante, para depurar un programa, observar cómo queda el archivo intermedio generado por el preprocesador después de todas las sustituciones, inclusiones, etc. La mayoría de los compiladores cuentan con una opción que permite generar este archivo intermedio y detener allí la compilación, para poder estudiarlo. Otra opción relacionada con el preprocesador que suelen ofrecer los compiladores es aquella que permite definir, al tiempo de la compilación y sin modificar los fuentes, símbolos que se pondrán a la vista del preprocesador. Así, la estructura final de un programa puede depender de decisiones tomadas al tiempo de compilación. Esto permite, por ejemplo, aumentar la portabilidad de los programas, o generar múltiples versiones de un sistema sin diseminar el conocimiento reunido en los módulos fuente que lo componen. Finalmente, aunque el compilador tiene un directorio default donde buscar los archivos de inclusión, es posible agregar otros directorios para cada compilación con argumentos especiales si es necesario. Ejercicios 1. Dé ejemplos de directivas de preprocesador: para incluir un archivo proporcionado por el compilador para incluir un archivo confeccionado por el usuario - 13 - Ejercicios Eduardo Grosclaude 2001 para definir una constante numérica para compilar un segmento de programa bajo la condición de estar definida una constante idem bajo la condición de ser no definida idem bajo la condición de que un símbolo valga una cierta constante idem bajo la condición de que dos símbolos sean equivalentes 2. Proponga un método para incluir un conjunto de archivos en un módulo fuente con una sola directiva de preprocesador. 3. ¿Cuál es el ámbito de una definición de preprocesador? Si defino un símbolo A en un fuente y lo compilo creando un módulo objeto algo.o, ¿puedo utilizar A desde otro fuente, sin declararlo, a condición de linkeditarlo con algo.o? 4. ¿Qué pasa si defino dos

veces el mismo símbolo en un mismo fuente? 5. Un cierto header A es incluido por otros headers B, C y D. El fuente E necesita incluir a B, C y D. Proponga un método para poder hacerlo sin obligar al preprocesador a leer el header A más de una vez. 6. Edite el programa hello.c del ejemplo del capítulo 1 reemplazando la cadena "Hola, mundo!"por un símbolo definido a nivel de preprocesador. 7. Edite el programa hello.c incluyendo la compilación condicional de la instrucción de impresión printf() sujeta a que esté definido un símbolo de preprocesador llamado IMPRIMIR. Compile y pruebe a) sin definir el símbolo IMPRIMIR, b) definiéndolo con una directiva de preprocesador, c) definiéndolo con una opción del compilador. ¿En qué casos es necesario recompilar el programa? 8. Escriba una macro que imprima su argumento usando la función printf(). Aplíquela para reescribir hello.c de modo que funcione igual que antes. 9. ¿Cuál es el resultado de preprocesar las líneas que siguen? Es decir, ¿qué recibe exactamente el compilador luego del preprocesado? define ALFA 8 define BETA 2*ALFA define PROMEDIO(x,y) (x+y)/2 a=ALFA*BETA; b=5; c=PROMEDIO(a,b); 10 ¿Qué está mal en los ejemplos que siguen? a) define PRECIO 27.5 PRECIO=27.7; b) define 3.14 PI c) define doble(x) 2*x; alfa=doble(6)+5; - 14 - Lenguaje C Ejercicios 11. Investigue la función de los símbolos predefinidos $_{STDC}$, $_{FILE}$ y $_{LINE}$.

Capítulo 3

Tipos de datos y expresiones

En general, las **expresiones** en C se construyen conectando, mediante **operadores**, diversos elementos, tales como **identificadores** de variables, **constantes** e invocaciones de **funciones**. Cada uno de estos elementos tiene un valor al tiempo de ejecución, y debe ocupar -al menos temporariamente, mientras se calcula el resultado de la expresión- un lugar en memoria. Al evaluar cada expresión, el compilador crea, para alojar cada subexpresión de las que la constituyen, **objetos de datos**, que pueden pensarse como espacio de memoria reservado temporariamente para contener valores. Al completar el cálculo de la expresión, el resultado nuevamente debe ser alojado en un objeto de datos propio. Estos espacios de memoria son de diferentes "tamaños" (cantidades de bits) de acuerdo al **tipo de dato** de la subexpresión.

Así, las expresiones y subexpresiones en C asumen siempre un **tipo de datos**: alguno de los tipos básicos del lenguaje, o uno definido por el usuario. Una expresión, según las necesidades, puede convertirse de un tipo a otro. El compilador hace esto a veces en forma **automática**. Otras veces, el programador fuerza una **conversión de tipo** para producir un determinado resultado.

3.1. Declaración de variables

Los **tipos básicos** son:

- `char` (un elemento del tamaño de un byte)
- `int` (un número entero con signo)
- `long` (un entero largo)
- `float` (un número en punto flotante)
- `double` (un número en punto flotante, doble precisión)

Cuando declaramos una variable o forzamos una conversión de tipo, utilizamos una **especificación de tipo**. Ejemplos de declaración de variables:

```
char a;
int alfa,beta;
float x1,x2;
```

Los **tipos enteros** (`char`, `int` y `long`) admiten los modificadores `signed` (con signo) y `unsigned` (sin signo). Un objeto de datos **unsigned** utiliza todos sus bits para representar magnitud; un **signed** utiliza un bit para signo, en representación complemento a dos.

El modificador `signed` sirve sobre todo para explicitar el signo de los chars. El default para un `int` es signed; el default para `char` puede ser signed o unsigned, dependiendo del compilador.

```
unsigned int edad;
signed char beta;
```

Un int puede afectarse con el modificador `short` (corto).

```
short i;
unsigned short k;
```

Cuando en una declaración aparece sólo el modificador unsigned o short, y no el tipo, **se asume int**. El tipo entero se supone el tipo básico manejable por el procesador, y es el tipo por omisión en varias otras situaciones. Por ejemplo, cuando no se especifica el **tipo del valor devuelto** por una función.

El modificador long puede aplicarse también a float y a double. Los tipos resultantes pueden tener más precisión que los no modificados.

```
long float e;
long double pi;
```

3.2. Tamaños de los objetos de datos

El lenguaje C no define el tamaño de los objetos de datos de un tipo determinado. Es decir, un entero puede ocupar 16 bits en una implementación del compilador, 32 en otra, o aun 64. Un long puede tener o no más bits que un int. Un short puede ser o no más corto que un int. Según K&R, lo único seguro es que "*un short no es mayor que un int, que a su vez no es mayor que long*".

Por supuesto, distintos tamaños en bits implican diferentes rangos de valores. Si deseamos **portar** un programa, hecho bajo una implementación del compilador, a otra, no es posible asegurar a priori el rango que tomará un tipo de datos. La fuente ideal para conocer los rangos de los diferentes tipos, en una implementación determinada, es -además del manual del compilador- el header `limits.h` de la Biblioteca Standard. Debe recordarse que cualquier suposición que hagamos sobre el rango o tamaño de un objeto de datos afecta la portabilidad de un programa en C.

Las siguientes líneas son parte de un archivo `limits.h` para una implementación en particular:

```
/* Minimum and maximum values a 'signed short int' can hold. */
#define SHRT_MIN      (-32768)
#define SHRT_MAX      32767
/* Maximum value an 'unsigned short int' can hold. (Minimum is 0.) */
#define USHRT_MAX     65535
/* Minimum and maximum values a 'signed int' can hold. */
#define INT_MIN (-INT_MAX - 1)
#define INT_MAX 2147483647
/* Maximum value an 'unsigned int' can hold. (Minimum is 0.) */
#ifdef __STDC__
#define UINT_MAX      4294967295U
#else
#define UINT_MAX      4294967295
#endif
/* Minimum and maximum values a 'signed long int' can hold. */
#ifdef __alpha__
#define LONG_MAX      9223372036854775807L
#else
#define LONG_MAX      2147483647L
#endif
#define LONG_MIN      (-LONG_MAX - 1L)
```

```
/* Maximum value an 'unsigned long int' can hold. (Minimum is 0.) */
# ifdef __alpha__
#define ULONG_MAX     18446744073709551615UL
#else
#ifdef __STDC__
#define ULONG_MAX     4294967295UL
#else
#define ULONG_MAX 4294967295L
#endif
#endif
```

Cuando una operación sobre una variable provoca overflow, no se obtiene ninguna indicación de error. El valor sufre **truncamiento** a la cantidad de bits que pueda alojar la variable.

Así, en una implementación donde los ints son de 16 bits, si tenemos en una variable entera el máximo valor positivo:

```
int a;
a=32767; /* a=0111111111111111 binario */
a=a+1;
```

Al calcular el nuevo valor de a, aparece un 1 en el bit más significativo, lo cual, según la representación de los enteros, lo transforma en un negativo (el menor negativo que soporta el tipo de datos, -32768).

Si el int es sin signo:

```
unsigned a;
a=65535; /* maximo valor de unsigned int */
a=a+1;
```

el incremento de a provoca overflow, y el nuevo valor de a se trunca a 16 bits, volviendo así a 0.

Siempre se puede saber el tamaño en bits de un tipo de datos aplicando el operador `sizeof()` a una variable o a la especificación de tipo.

3.3. Operaciones con distintos tipos

En una expresión en C pueden aparecer componentes de diferentes tipos. Durante la evaluación de una expresión cuyas subexpresiones sean de **tipos diferentes**, deberá tener lugar una **conversión**, ya sea implícita o explícita, para llevar ambos operandos a un **tipo de datos común** con el que se pueda operar. La forma en que el compilador resuelve las conversiones implícitas a veces provoca algunas sorpresas.

Truncamiento en asignaciones

Para empezar, una asignación de una expresión de un tipo dado a una variable de un tipo "menor" (en el sentido del tamaño en bits de cada objeto de datos), no sólo es permitida en C, sino que la conversión se hace en forma automática y generalmente sin ningún mensaje de tiempo de compilación ni de ejecución. Por ejemplo,

```
int a;
float b;
...
a=b;
```

En esta asignación tenemos miembros de diferentes tamaños. El resultado en a será el truncamiento del valor entero de b a la cantidad de bits que permita un int. Es decir, se tomará la parte entera de b,

y de ese valor se copiarán en el objeto de datos de a tantos bits como quepan en un int, tomándose los menos significativos.

Si el valor de b es, por ejemplo, 20.5, a terminará valiendo 20, lo que es similar a aplicar una función "parte entera" implícitamente, y no demasiado incongruente. Pero si la parte entera de b excede el rango de un entero (por ejemplo si b=99232.5 en una plataforma donde los enteros son de 16 bits), el resultado en a no tendrá lógica aparente. En el primer caso, los bits menos significativos de b que "caben" en a conservan el valor de b; en el segundo caso, no.

En la sentencia:

```
a=19.27 * b;
```

a contendrá los `sizeof(int)` bits menos significativos del resultado de evaluar la expresión de la derecha, truncada sin decimales.

Promoción automática de expresiones

Por otra parte, se tienen las reglas de promoción automática de expresiones. Enunciadas en forma aproximada (luego las daremos con más precisión), estas reglas dicen que el compilador hará estrictamente las conversiones necesarias para llevar todos los operandos al tipo del "mayor" entre ellos. El resultado de evaluar una operación aritmética será del tipo del "mayor" de sus operandos.

A veces, esto no es lo que se desea. Por ejemplo, dada la sentencia:

```
a=3/2;
```

se tiene que tanto la constante 3 como la constante 2 son vistas por el compilador como ints; el resultado de la división será también un entero (la parte entera de 3/2, o sea 1). Aun más llamativo es el hecho de que si declaramos previamente:

```
float a;
```

el resultado es casi el mismo: a terminará conteniendo el valor float 1.0, porque el problema de truncamiento se produce ya en la evaluación del miembro derecho de la asignación.

Operador cast

En el ejemplo anterior, ¿cómo recuperar el valor correcto, con decimales, de la división? Declarar a la variable a como **float** es necesario, pero no suficiente. Para que la expresión del miembro derecho sea **float** es necesario que **al menos uno** de sus operandos sea **float**. Hay dos formas de lograr esto; la primera es escribir cualquiera de las subexpresiones como constante en punto flotante (con punto decimal, o en notación exponencial):

```
a=3./2;
```

La segunda consiste en forzar explícitamente una conversión de tipo, con un importante operador llamado **cast**, de la siguiente manera.

```
a=(float)3/2;
```

El operador **cast** es la aclaración, entre paréntesis, del tipo al cual queremos convertir la expresión (en este caso, la subexpresión **3**). Da lo mismo aplicarlo a cualquiera de las constantes. Sin embargo, lo siguiente **no funcionará**:

```
a=(float)(3/2);
```

Aquí el operador **cast** se aplica a la expresión **ya evaluada como entero**, con lo que volvemos a tener un valor **1.0**, float, en a.

Reglas de promoción en expresiones

Son aplicadas por el compilador en el orden que se da más abajo (tomado de K&R, 2a. ed.). Ésta es una lista muy detallada de las comprobaciones y conversiones que tienen lugar. Para la mayoría de los propósitos prácticos, basta tener en cuenta la regla de **llevar ambos operandos al tipo del "mayor"** de ellos.

Entendemos por "promoción entera" el acto de llevar los objetos de tipo `char`, `enum` y **campos de bits** a **int**; o, si los bits de un int no alcanzan a representarlo, a **unsigned int**.

1. Si cualquier operando es long double, se convierte el otro a long double.
2. Si no, si cualquier operando es double, se convierte el otro a double.
3. Si no, si cualquier operando es float, se convierte el otro a float.
4. Si no, se realiza promoción entera sobre ambos operandos.
5. Si cualquiera de ellos es unsigned long int, se convierte el otro a unsigned long int.
6. Si un operando es long int y el otro es unsigned int, el efecto depende de si un long int puede representar a todos los valores de un unsigned int.
7. Si es así, el unsigned int es convertido a long int.
8. Si no, ambos se convierten a unsigned long int.
9. Si no, si cualquier operando es long int, se convierte el otro a long int.
10. Si no, si cualquier operando es unsigned int, se convierte el otro a unsigned int.
11. Si no, ambos operandos son int.

3.4. Observaciones

Nótese que **no existen** tipos `boolean` ni `string`. Más adelante veremos cómo manejar datos de estas clases.

El tipo **char**, pese a su nombre, no está restringido a la representación de caracteres. Por el contrario, un char **tiene entidad aritmética**. Almacena una cantidad **numérica** y puede intervenir en operaciones matemáticas. En determinadas circunstancias, y sin perder estas propiedades, puede ser interpretado como un carácter (el **carácter cuyo código ASCII contiene**).

En general, en C es conveniente habituarse a pensar en los datos separando la **representación** (la forma como se almacena un objeto) de la **presentación** (la forma como se visualiza). Un mismo patrón de bits almacenado en un objeto de datos puede ser visto como un número decimal, un carácter, un número hexadecimal, octal, etc. La verdadera naturaleza del dato es la representación de máquina, el patrón de bits almacenado en el objeto de datos.

%d	entero, decimal
%u	entero sin signo, decimal
%l	long, decimal
%c	carácter
%s	cadena
%f	float
%lf	double
%x	entero hexadecimal

Cuadro 3.1: Especificaciones de conversión de printf().

3.5. Una herramienta: printf()

Con el objeto de facilitar la práctica, describimos aquí la función de Biblioteca Standard `printf()`.

- La función de salida `printf()` lleva un **número variable de argumentos**.
- Su primer argumento siempre es una cadena o constante string, la **cadena de formato**, conteniendo texto que será impreso, más, opcionalmente, **especificaciones de conversión**.
- Las especificaciones de conversión comienzan con un signo "%". Todo otro conjunto de caracteres en la cadena de formato será impreso textualmente.
- Cada especificación de conversión determina la manera en que se imprimirán los restantes argumentos de la función.
- Deben existir tantas especificaciones de conversión como argumentos luego de la cadena de formato.
- Un mismo argumento de un tipo dado puede ser impreso o presentado de diferentes maneras según la especificación de conversión que le corresponda en la cadena de formato (de aquí la importancia de separar representación de presentación)
- Las especificaciones de conversión pueden estar afectadas por varios **modificadores** opcionales que determinan, por ejemplo, el ancho del campo sobre el cual se escribirá el argumento, la cantidad de decimales de un número, etc.
- Las principales especificaciones de conversión están dadas en el Cuadro 3.1.

Ejemplos

Este programa escribe algunos valores con dos especificaciones de formato distintas.

```
main() {
        int i,j;
        for(i=65, j=1; i<70; i++, j++)
                printf("vuelta no. %d: i=%d, i=%c\n",j,i,i);
}
```

Salida del programa:

```
vuelta no. 1: i=65, i=A
vuelta no. 2: i=66, i=B
vuelta no. 3: i=67, i=C
vuelta no. 4: i=68, i=D
vuelta no. 5: i=69, i=E
```

El programa siguiente escribe el mismo valor en doble precisión pero con diferentes modificadores del campo correspondiente, para incluir una cierta cantidad de decimales o alinear la impresión.

```
main() {
       double d;
       d=3.141519/2.71728182;
       printf("d=%lf\n",d);
       printf("d=%20lf\n",d);
       printf("d=%20.10lf\n",d);
       printf("d=%20.5lf\n",d);
       printf("d=%.10lf\n",d);
}
```

Salida:

```
d=1.156126
d=           1.156126
d=       1.1561255726
d=            1.15613
d=1.1561255726
```

3.6. Ejercicios

1. ¿Cuáles de entre estas declaraciones contienen errores?

   a. integer a;
   b. short i,j,k;
   c. long float (h);
   d. double long d3;
   e. unsigned float n;
   f. char 2j;
   g. int MY;
   h. float ancho, alto, long;
   i. bool i;

2. Dé declaraciones de variables con tipos de datos adecuados para almacenar:

   a. La edad de una persona.
   b. Un número de DNI.
   c. La distancia, en Km, entre dos puntos cualesquiera del globo terrestre.
   d. El precio de un artículo doméstico.
   e. El valor de la constante PI expresada con 20 decimales.

3. Prepare un programa con variables conteniendo los valores máximos de cada tipo entero, para comprobar el resultado de incrementarlas en una unidad. Imprima los valores de cada variable antes y después del incremento. Incluya **unsigneds**.

4. Lo mismo, pero dando a las variables los valores mínimos posibles, e imprimiéndolas antes y después de decrementarlas en una unidad.

5. Averigüe los tamaños de todos los tipos básicos en su sistema aplicando el operador sizeof().

6. Si se asigna la expresión (3-5) a un unsigned short, ¿cuál es el resultado? ¿Depende de qué formato de conversión utilicemos para imprimirlo?

7. ¿Qué hace falta corregir para que la variable `r` contenga la división exacta de `a` y `b`?

```
int a, b;
float r;
a = 5;
b = 2;
r = a / b;
```

8. ¿Qué resultado puede esperarse del siguiente fragmento de código?

```
int a, b, c, d;
a = 1;
b = 2;
c = a / b;
d = a / c;
```

9. ¿Cuál es el resultado del siguiente fragmento de código? Anticipe su respuesta en base a lo dicho en esta unidad y luego confírmela mediante un programa.

```
printf("%d\n", 20/3);
printf("%f\n", 20/3);
printf("%f\n", 20/3.);
printf("%d\n", 10%3);
printf("%d\n", 3.1416);
printf("%f\n", (double)20/3);
printf("%f\n", (int)3.1416);
printf("%d\n", (int)3.1416);
```

10. Escribir un programa que multiplique e imprima $100000 * 100000$. ¿De qué tamaño son los ints en su sistema?

11. Convertir una moneda a otra sabiendo el valor de cambio. Dar el valor a dos decimales.

12. Escriba y corra un programa que permita saber si los chars en su sistema son signed o unsigned.

13. Escriba y corra un programa que asigne el valor 255 a un char, a un unsigned char y a un signed char, y muestre los valores almacenados. Repita la experiencia con el valor `-1` y luego con `'\377'`. Explicar el resultado.

14. Copiar y compilar el siguiente programa. Explicar el resultado.

```
main() {
        double x;
        int i;
        i = 1400;
        x = i; /* conversion de int a double */
        printf("x = %10.6le\ti = %d\n",x,i);
        x = 14.999;
        i = x; /* conversion de double a int */
        printf("x = %10.6le\ti = %d\n",x,i);
        x = 1.0e+60;
        i = x;
        printf("x = %10.6le\ti = %d\n",x,i);
}
```

15. Escriba un programa que analice la variable v conteniendo el valor 347 y produzca la salida:

```
3 centenas
4 decenas
7 unidades
```

    (y, por supuesto, salidas acordes si v toma otros valores).

16. Sumando los dígitos de un entero escrito en notación decimal se puede averiguar si es divisible por 3 (se constata si la suma de los dígitos lo es). ¿Esto vale para números escritos en otras bases? ¿Cómo se puede averiguar esto?

17. Indicar el resultado final de los siguientes cálculos

    a. int a; float b = 12.2; a = b;
    b. int a, b; a = 9; b = 2; a /= b;
    c. long a, b; a = 9; b = 2; a /= b;
    d. float a; int b, c; b = 9; c = 2; a = b/c;
    e. float a; int b, c; b = 9; c = 2; a = (float)(b/c);
    f. float a; int b, c; b = 9; c = 2; a = (float)b/c;
    g. short a, b, c; b = -2; c = 3; a = b * c;
    h. short a, b, c; b = -2; c = 3; a = (unsigned)b * c;

18. Aplicar operador cast donde sea necesario para obtener resultados apropiados:

    a.

```
int a; long b; float c;
a = 1; b = 2; c = a / b;
```

    b.

```
long a;
int b,c;
b = 1000; c = 1000;
a = b * c;
```