Breve introducción al lenguaje C

Marcelo Arroyo

2014

Resumen

El lenguaje C sigue siendo el lenguaje mas utilizado para la implementación de programas de sistemas, aplicaciones en sistemas *embedded*, microcontroladores, DSPs y en cualquier aplicación que requiera un pequeño *fooprint* y máxima performance.

A pesar de ser un lenguaje simple, con mecanismos básicos, muchos estudiantes y programadores lo consideran un lenguaje intimidante y propenso a cometer errores.

Esto se debe a que no hay mucha bibliografía consisa, clara y que se enfoque en las técnicas y estilos simples de programación y la adecuada utilización de los mecanismos que ofrece el lenguaje.

Este documento pretende describir las principales características del lenguaje y estrategias de uso de esos mecanismos para lograr comprender código existente y escribir programas legibles, eficientes y bien estructurados.

1. Introducción

El leguaje fue creado por Dennis Ritchie en 1972 en los laboratorios `Bell`. Sus predecesores fueron los lenguajes `B` y `BCPL`. Originalmente fue diseñado para la implementación del sistema operativo `UNIX`.

C es un lenguaje de bajo nivel, es decir que está pensado para manipular objetos que son o pueden ser fácilmente manejables por el hardware, como ser valores numéricos y direcciones de memoria.

No proporciona operadores para definir y manupular objetos compuestos como cadenas de caracteres (strings), listas, arreglos o conjuntos. Tampoco provee operadores o notaciones específicas para hacer entrada-salida ni

manejo de *heap*[1].

Obviamente, todas estas abstracciones pueden definirse o usarse desde *bibliotecas*[2] existentes.

La *biblioteca estándar*, que viene acompañada con cualquier compilador moderno de C, provee un gran conjunto de facilidades de uso común como por ejemplo, para entrada-salida (`fopen(), fclose(), fread(), fwrite(), ...`), variables *heap* (`malloc()`, `free()`), manipulación de arreglos y strings (`strcpy(), memcpy(), strcat(), ...`) y muchas otras.

El compilador C generalmente incluye un *pre-procesador* el cual se encarga de procesar directivas generalmente incluidas en un programa. En rigor el pre-procesador es una herramienta externa de utilidad pero que no tiene nada que ver con el lenguaje en sí mismo. De hecho es posible utilizar el pre-procesador de C para procesar las directivas en otro tipos de archivos de texto.

En 1989 fue estandarizado por primera vez. En 1999 se publicó la segunda edición del estándar[2], el cual incluye algunas modificaciones del lenguaje, principalmente en la verificación de tipos de datos realizados por el compilador, *variadic macros* soportadas por el pre-procesador, etc.

El lenguaje ha ido evolucionando y el último estándar es de 2011[2].

En este documento, se describe el lenguaje descripto en el último estándar.

2. Estructura de un programa C

Un programa C es una secuencia de *declaraciones y/o definiciones.*

Una *declaración* expresa la existencia de un tipo de datos, una variable o constante o una función.

El siguiente ejemplo muestra la estructura básica de un programa C.

```
int g = 100; /* global g (inicializada) */
unsigned long int lg; /* global (no inicializada) */

/* f : int -> int */
int f(int x)
{
        long int result; /* variable local (no inicializada) */
```

[1]El *heap* es una área de memoria para almacenar valores creados dinámicamente (ej: operadores `new` y `dispose` de Pascal).

[2]Las cuales son módulos que proveen implementaciones de esas abstracciones.

```
        result = x*g;
        return result;
}

/* Funcion principal: sin argumentos, retorna un entero */
int main(void)
{
        lg = f(g);        /* lg <- 100*100 */
        return 0;    /* codigo de salida exitoso */
}
```

En el ejemplo anterior podemos ver las declaraciones (en realidad definiciones) de las variables globales[3] `g` y `lg`. Ambas pueden contener valores numéricos de tipo entero. El tipo `long int` incrementa el número de bits usados para representar un entero.

Cualquier texto entre `/*` y `*/` es un comentario (es ignorado por el compilador).

Las declaraciones `f` y `main` definen las funciones `f` y `main`, respectivamente. Una función `f` se define mediante la siguiente sintaxis: `return_type f (type1 arg1, type2 arg2, ...) { body }`

donde `return_type` es el tipo del valor retornado (computado) por `f`, y `arg1, arg2, ...` son los *parámetros formales* de la función, es decir sus valores de entrada (inputs).

El *cuerpo de la función* es una *sentencia de bloque*.

Un *bloque* es una secuencia de declaraciones de variables o constantes, seguida de sentencias.

Una *sentencia* es una *expresión* o una *sentencia de control*.

El compilador tratará de generar un ejecutable el cual comenzará por la función `main()`. Esta función no es especial, ya que es una función como cualquier otra, sólo que el compilador está configurado para que *main* sea el símbolo designado como el *punto de entrada* del programa.

3. Tipos de datos

3.1. Básicos

Los tipos de datos básicos provistos por C son los siguientes:

[3]Una variable global tiene tiempo de vida igual a la de la ejecución del programa.

Tipo	Descripción
`char`	Unidad de datos mínima direccionable (byte)
`int`	Número entero
`float`	Real en punto flotante
`double`	Real en punto flotante (doble precisión)

Estos tipos de datos pueden declararse usando un *modificador* de representación:

Modificador	Descripción
`long`	Incrementa el número de bits
`signed`	Con representación del signo (default)
`unsigned`	Sin signo
`const`	No modificable (sólo inicializable)

Además de estos tipos básicos existe el tipo `void`, el cual representa *cualquier cosa* o *nada*, dependiendo del contexto.

Una declaración de una función con tipo re retorno `void` significa que no retorna ningún valor, es decir que la función representa un *procedimiento*.

Si en la lista de parámetros formales se usa `void`, significa que la función no toma parámetros.

Más adelante se verá que un puntero a algo de tipo `void` representa una dirección de memoria de un valor de cualquier tipo. Esto generalmente se usa para implementar *polimorfismo*[4].

Para cada tipo básico, existen notaciones para representar valores literales. Los valores de tipo entero se denotan en decimal de forma natural. Ejemplos: 15, -1, 0.

Para los valores numéricos, existen prefijos y sufijos para denotar la base y los modificadores de representación. Los siguientes prefijos se usan para denotar diferentes bases de representación: `0` para valores en octal, texttt0x para valores en hexadecimal, texttt0b para valores en binario (gcc extension).

Los sufijos describen el tamaño y convención de la reprentación y pueden ser `L` o `l` (long) y `U` o `u` (unsigned).

Ejemplos:

```
long int mem_addr = 0xFF0AL;
unsigned char perm = 0744;
...
mem_addr += 0xF0BCu;
```

[4]Operaciones sobre valores u objetos con *diferentes formas*.

Es notable que C carece del tipo de valores lógicos (booleans). En realidad una expresión que evalúa a cero se asume como `false`, sino como `true`.

3.2. Enumeraciones

A veces es necesario nombrar valores o simplemente representar valores simbólicos o constantes nombradas.

Ejemplo:

```
enum dia_semana {dom,lun,mar,mie,jue,vie,sab};
```

Define un tipo de datos `dia_semana` cuyos valores posibles (dominio) son `dom,lun,...,vie,sab`.

Estos valores se representarán como los enteros 0 (dom),1 (lun), ..., 6(sab) y son valores constantes (es decir, la siguiente expresión no es válida).

```
lun = 5; /* Error */
```

Es posible definir los valores de representación a cada constante simbólica. La declaración anterior es equivalente a:

```
enum dia_semana {dom=0,lun=1,mar=2,mie=3,jue=4,vie=5,sab=6};
```

Esto define el tipo `dia_semana`. Una declaración de la forma

```
dia_semana day = mar;
```

define la variable `day` inicializada con el valor `mar`.

Internamente, los valores posibles de los símbolos de una enumeración se representan como enteros, por lo que es posible realizar su conversión.

3.3. Punteros

Un puntero representa una dirección de memoria de un valor u objeto. Los punteros en C++ son tipados, es decir que un puntero puede tomar direcciones de objetos o valores del tipo de su declaración[5].

Una variable puntero a valores de tipo `T` se declara de la forma: `T * ptr;`

Los punteros son tipos básicos del lenguaje por lo que se pueden asignar, comparar y usar en expresiones artiméticas, aunque ésta es levemente diferente a la aritmética normal de enteros.

A modo de ejemplo, la siguiente expresión es válida:

```
int v1=1,v2=2,x;
int * ptr = &v1;

x = ++ptr;
```

[5]Aunque es posible violar esta restricción usando *casts*.

La variable puntero `ptr` contiene la dirección de memoria donde se almacena el contenido de `v1`. El operador $\& v$ toma la dirección de memoria donde se almacena el valor `v`.

La expresión `++ptr` incrementa la dirección de memoria contenida en `ptr` en $1 \times sizeof(int)$. En una arquitectura de 32 bits, comúnmente el `sizeof(int)` retorna 4. Es muy probable que en el ejemplo anterior, en la variable `x` se copie el valor 2 (contenido de `v2`), ya que `v2` está almacenada a continuación en la memoria de la variable `v1`.

A un puntero puede asignarse un valor numérico sin signo, el cual representará una dirección de memoria. Ejemplo: `int * ptr = 0xFA01;`

La constante `NULL`, definida en la biblioteca estándar, representa una dirección inválida, es decir que su referenciación será capturada por el sistema como una excepción y generará la finalización del programa[6].

3.4. Constantes

Es posible definir contantes anteponiendo el modificador *const* en la declaración.

Ejemplo: `const float pi = 3.141516;`

Una declaración de una constante requiere la inicialización y no podrá ser modificada a posteriori.

3.5. Arreglos

Un arreglo (unidimensional) en C es una secuencia de valores almacenados en forma contigua en memoria.

Un arreglo `v` de dimensión `N` (N valores) de tipo `t` se define como:
$t\ v[N];$
El elemento i-ésimo se denota con la expressión $v[i]$, $(0 \leq i \leq N-1)$.

Un arreglo puede inicializarse en su definición y en ese caso puede omitirse la dimensión y el compilador la inferirá desde la expresión de inicialización. Ejemplos:

```
int v1[N];              /* unintialized array */
int v2[9] = {1,2};      /* v2[0]=1, v2[1]=2, others not initialized */
int v3[] = {0,1,2,3}; /* vector of dimention 4 */
```

[6]En sistemas tipo UNIX generará una `segmentatio fault`.

C utiliza una notación especial (literal agregado) para describir arreglos de caracteres. Ejemplo:

```
char s[5] = {'h','o','l','a',0};
int r = strcmp("hola",s); /* r==0, s1 and s2 have equal contents */
```

Desde el ejemplo anterior puede inferirse que el literal conteniendo la palabra `hola` se representa como un arreglo de 5 caracters (incluye la marca final, el valor cero).

C no sabe asignar (copiar) arreglos. Esto se debe hacer con la función de biblioteca `strcpy()`. En el ejemplo anterior, la función de biblioteca `strcmp` está definida como:

```
/* returns 0 if s1 == s2, >0 if s1 > s2, <0 otherwise */
int strcmp(const char * s1, const char s2[])
{
        int i;

        for(i=0; s1[i] != 0; i++) {
          if (s1[i] < s2[i])
            return -i;
          if (s1[i] > s2[i])
            return i;
        }
        return 0;
}
```

El ejemplo anterior muestra que cuando se pasa un arreglo como parámetro, es decir que el parámetro actual es una expresión igual al identificador del arreglo, éste evalúa a un puntero del tipo base correspondiente, en realidad a la dirección de memoria del primer elemento. Por lo tanto, en la función, el parámetro formal correspondiente deberá declararse como un puntero o usando la notación de arreglo

explicitando su dimensión si se conoce en tiempo de compilación u omitiéndola si no se conoce.

Es conveniente usar la notación de arreglo en los parámetros formales ya que es una notación mas clara de que lo que se espera es un arreglo y no sólo un puntero a un elemento.

En el caso que una función reciba como parámetros un arreglo de dimensión desconocida en tiempo de compilación, su dimensión puede omitirse, en cuyo caso generalmente se define un parámetro extra para recibir la dimen-

sión actual en cada invocación.

Un arreglo puede recorrerse utilizando el operador de indexación (corchetes) o usando aritmética de punteros como en el siguiente ejemplo:

```
int v[4] = {0,1,2,3};
int *p, r;

r = v[1]; /* r == 1 */
r = *v+2; /* r == 2 == v[2] */
p = v;    /* p == &v[0] */
r = *p;   /* r == v[0] == 0 */
```

C sólo permite arreglos unidimensionales (vectores), pero es simple definir matrices (arreglos bi-dimensionales) o de mayores dimensiones declarando vectores de vectores.

A modo de ejemplo, el siguiente programa define y opera sobre matrices.

```
#include <stdio.h>

#define N 10
#define M  5

void matrix_init(int m[N][M])
{
        int i,j;

        for(i=0; i<N; i++)
                for(j=0; j<N; j++)
                        m[i][j] = i*j;
}

int matrix_get_element(int (*m)[M], int i, int j)
{
        return m[i][j];
}

void matrix_set_element(int m[][M], int i, int j, int value)
{
       m[i][j] = value;
}

int main(void)
{
        int matrix[N][M];

        matrix_init(matrix);
        printf("Element (i,j) = %d\n", matrix_get_element(matrix,3,4));
```

```
        return 0;
}
```

Notar las definiciones de los paramétros formales de las funciones. Son tres formas correctas alternativas. En `matrix_get_element`: es un puntero a vectores de dimensión `M`. En `matrix_set_element` se usa la notación de arreglo omitiendo la dimensión. En `matrix_init` está declarado igual que la declaración de su parámetro actual.

Se debe notar que en el caso de arreglos de arreglos, es posible omitir la primera dimensión pero no las subsiguientes.

En el nuevo C estándar siempre es preferible la notación de arreglo (aún cuando las dimensiones sean desconocida) ya que es posible declarar el parámetro de la siguiente forma:

```
T1 f(int n, int m, T2 array[n][m]);
```

Por lo anterior, la forma de declaración `char * str` para los *strings* es obsoleta, deberían declararse como `char str[]` o `char str[d]`.

3.6. Estructuras

Una estructura representa un producto cartesiano (o registro) de valores. Cada valor que forma parte de una estructura o registro se denomina *campo*).

Un tipo registro se define en C de la siguiente forma:

```
struct person {
  int id;
  char name[31];
  char address[41];
};
```

Una variable de tipo `person` se define como:

```
struct person p;
```

El acceso a cada campo se realiza mediante el operador .:

```
p.id = 4567;
strcpy(p.name,"John T.");
```

3.7. Uniones

Una unión, tambié conodida como registro variante (o variant) en otros lenguajes, permite encapsular en una estructura diferentes subestructuras

que comparten el área de memoria.

Por ejemplo, supongamos que queremos tener dos visiones diferentes de un entero (de 32 bits):

```
typedef unsigned char byte;
union {
        int value;
        byte bytes[4];
} n;
n.value = 1024 + 256 + 1;
byte b = n.bytes[0]; /* less significant byte? (yes on i386) */
n.bytes[0] += 4;     /* on i386, n.value should be 1292 */
```

Un valor de tipo `figure` tiene el campo `type` común y el campo `shape` es variante: puede contener el valor de un rectángulo, triángulo o círculo, pero sólo uno de ellos en un momento dado.

4. Declaraciones y definiciones

Una *declaración* expresa la existencia de una entidad (tipo, variable, constante o función).

Una definición introduce un nuevo elemento en el sistema (tipo, variable, constante o función).

Así, una declaración de la forma:

```
const double pi = 3.141516;
```

define una constante llamada `pi` (de tipo `double`) con un valor dado.

En cambio la declaración

```
double area\_square(square x);
```

expresa la existencia de la función `area_square` la cual puede estar definida mas abajo en el texto del programa (forward declaration).

Este tipo de declaraciones generalmente se expresan para que el compilador pueda conocer su perfil antes de procesar su definición para que pueda realizar el chequeo de tipos correspondiente (por ejemplo, en una invocación).

Una definición generalmente genera un requerimiento de memoria.

En una declaración (en realidad, definición) de variables (o constantes) podemos expresar su valor inicial. Ej:

```
char c = 'A'; int x = 0, y; /* y no inicializada */
```

En C declaramos *enumeraciones*, existencia de funciones o variables y nuevos tipos[7].

Un ejemplo de cómo definir un nuevo tipo:

```
typedef unsigned long int big_number;
```

Una declaración (definición) de la forma:

```
big_number n;
```

es equivalente a:

```
unsigned long int n;
```

5. Expresiones

Una expresión representa un valor (número, dirección de memoria, estructura, etc).

Las expresiones mas simples se forman a partir de valores constantes (literales) como `15` o `"hola"`, constantes simbólicas y variables.

La tabla 5 muestra algunos ejemplos de valores literales constantes.

Literal	**tipo**
20, -5, 0, ...	Enteros
0L, 0755, 0x20AD	Enteros; long, octal, hexadecimal
3.1415, .5, -8.23	Reales en punto flotante
123.455E-3	$123{,}455 \times 10^{-3}$
'A', '\n', '\t', '\65'	char (ascii codes)
"Hola"	String (arreglo de chars)

Figura 1: Literales

La notación de strings (agregado) como la última línea de la tabla 5 es simplemente una abreviación (syntactic sugar) de

```
const char anon[5] = {'H','o','l','a',0};
```

por lo que podemos apreciar que los strings se representan como arreglos de caracteres con marca final (el valor 0).

[7]En realidad definimos nuevos nombres (alias) a tipos existentes.

Los operadores, que se aplican a expresiones, permiten formar expresiones complejas.

La tabla 5 describe los operadores y sus niveles de precedencia y asociatividad[8].

Algunos operadores requieren algo de atención. La asignación, por ejemplo, es una expresión, es decir que denota un valor[9]. Esto significa que las asignaciones pueden encadenarse. Por ejemplo:

```
x = y = z = 0;
```

Un error común es confundir el operador de chequeo de igualdad == con el de asignación. Por ejemplo:

```
r = (x = 0)? x+1 : -1; /* valid expression */
```

Este tipo de errores es tan común que muchos compiladores de C arrojan un aviso (warning).

Ejercicio: ¿a qué valor evalúa la expresión anterior?

Otro operador que es necesario analizar es el de invocación a función, el cual se denota como (). Una función $f : (x_1 \times x_2 \ldots x_n) \rightarrow R$, donde los argumentos (parámetros formales) son de tipo $T_1, T_2 \ldots, T_n$ y que retorna (computa) un valor de tipo R, se define como:

```
R f(T1 x1, T2 x2 ..., Tn xn)
{
        /* body */
}
```

y en una expresión se invoca de la forma $f(e1, e2 \ldots, e_n)$, donde $e1, e2 \ldots, e_n$ son expresiones llamados *parámetros actuales* (que evalúan a valores) de tipo $T1, T2 \ldots, Tn$, respectivamente.

Los parámetros se *copian*, es decir que se transmiten copias de los valores computados para cada expresión.

6. Ambientes y bloques

Un *bloque*, denotado por los símbolos { y }, agrupa una secuencia de declaraciones y/o sentencias (expresiones, sentencias de control, y otros blo-

[8]Estas características determinan el orden de evaluación de una expresión.

[9]En otros lenguajes, como Pascal, es un comando.

Operador	**Descripción**	**Asoc.**	**Prec.**
++	Incremento prefijo	izq. a der.	1
−−	Decremento prefijo	izq. a der.	1
()	LLamado a función y subexpr.	izq. a der.	1
[]	Indexado en arreglo	izq. a der.	1
− >	Acceso a campos desde puntero	izq. a der.	1
.	Acceso a campo de estructura	izq. a der.	1
!	Decremento prefijo	der. a izq.	2
	Complemento a 1	der. a izq.	2
−	Negación (unario)	der. a izq.	2
+	Positivo (unario)	der. a izq.	2
(t)	Cast (cambio de tipo)	der. a izq.	2
∗	Desreferencia de puntero	der. a izq.	2
&	Dirección de variable	der. a izq.	2
sizeof	Tamaño de rep. de dato	der. a izq.	2
∗	Multiplicación	izq. a der.	3
/	División	izq. a der.	3
%	Módulo (resto)	izq. a der.	3
+	Suma	izq. a der.	4
−	Resta	izq. a der.	4
<<	Desplazamiento de bits (a der.)	izq. a der.	5
>>	Desplazamiento de bits (a der.)	izq. a der.	5
<	Menor que	izq. a der.	6
<=	Menor o igual que	izq. a der.	6
>	Mayor que	izq. a der.	6
>=	Mayor o igual que	izq. a der.	6
==	Igual que	izq. a der.	7
! =	Igual que	izq. a der.	7
&	And (bit a bit)	izq. a der.	8
ˆ	Exclusive Or (XOR bit a bit)	izq. a der.	9
‖	Or (bit a bit)	izq. a der.	10
&&	And (lógico)	izq. a der.	11
‖‖	Or (lógico)	izq. a der.	12
? :	Expresión condicional	der. a izq.	13
=	Asignación	der. a izq.	14
+ =	Asignación acumulada	der. a izq.	14
− =	Asignación con resta	der. a izq.	14
∗ =	Asignación con multiplicación	der. a izq.	14
/ =	Asignación con división	der. a izq.	14
% =	Asignación con resto	der. a izq.	14
<<=	Asignación con dezp. a izq.	der. a izq.	14
>>=	Asignación con dezp. a der.	der. a izq.	14
& =	Asignación con and bit a bit	der. a izq.	14
ˆ =	Asignación con XOR bit a bit	der. a izq.	14
‖ =	Asignación con or bit a bit	der. a izq.	14
,	Secuencia (sep. de listas)	izq. a der.	15
++	Incremento posfijo	izq. a der.	15
−−	Decremento posfijo	izq. a der.	15

Figura 2: Operadores

ques).

Si bien el último estándar de C permite que en un bloque puedan mezclarse declaraciones y sentencias, muchos programadores utilizan el siguiente estilo:

```
{
        int x,y;                        /* Declaraciones */
        const char * s = "Hola";

        x = s[0];                       /* sentencias y/o expresiones */
        /*...*/
}
```

ya que las versiones anteriores de C requerían esta sintaxis.

Un bloque, además, define un nuevo ambiente de referencialidad. Las identificadores declarados en un bloque sólo son visibles en ese bloque (ambiente local).

Una declaración en un bloque puede *ocultar* a otros símbolos fuera del bloque, como en el siguiente ejemplo:

```
float x; /* global x */
...
        {
                int x; /* local variable */
                ...
                p(x);  /* use local x as parameter */
        }
```

C no permite definir funciones dentro de funciones[10].

6.1. Tiempo de vida y visibilidad de de los identificadores

El tiempo de vida de una variable global es el mismo que la duración de la ejecución del mismo. es decir que se crean (y eventualmente se inicializan) al momento del inicio del proceso (carga del programa) hasta que finaliza. Estas variables se almacenan en un área de memoria fijada por el compilador (área *estática*).

La visibilidad de una variable global es en todo el texto del programa, excepto que sea ocultada por una declaración de una variable local o pará-

[10]Otros lenguajes, como Pascal, por ejemplo, lo permiten.

metro formal con el mismo identificador.

El tiempo de vida de una variable local (o parámetro formal) declarado en una función es igual al tiempo de su *activación*, es decir que al retornar, éstas desaparecen. Por este motivo las variables locales y valores de los parámetros formales se denominan variables *automáticas* y se almacenan en la pila.

Una excepción a esta regla es si se declara con el modificador `static`, el cual afecta a su tiempo de vida, es decir que su valor permanece a través de sucesivas invocaciones[11].

La visibilidad de un identificador local está limitado al bloque en que fue definido.

El modificador `static` aplicado a una declaración global (variable o función), hace que su visibilidad o alcance quede limitada al archivo fuente que la define y no será visible desde otro archivo fuente (módulo). Generalmente se usa para *ocultar* información, como detalles de representación de datos y funciones auxiliares que no forma parten de la interface de un módulo.

Cabe aclarar que una variable `estática` local a una función su inicialización se realizará al comienzo de la ejecución del programa y no en cada invocación a la función, como sucede con las variables automáticas.

7. Sentencias de control

C contiene pocas sentencias de control de flujo de ejecución. Un bloque define una *secuencia* de declaraciones y comandos (expresiones y/o sentencias de control).

Sintácticamente, cada sentencia finaliza con el símbolo *;*.

7.1. Saltos y rótulos

La sentencia `goto L;` produce un salto en la ejecución secuencial normal a la sentencia precedida por el *rótulo (label)* `L:`. Ejemplo:

```
    if ( ! sucess )
      goto end;
    ... /* otras sentencias */
end:
    return error;
```

[11] Se almacenan en la misma área *estática* que las variables globales.

Un rótulo debe estar dentro de la misma función que el `goto` correspondiente.

7.2. Selección

En C existe un operador (expresión) condicional`(cond)? e1: e2`. Ejemplo:

```
r = (x>0)? x+1 : -1;
```

El ejemplo anterior podría escribirse como

```
if ( x > 0 )
  r = x+1;      /* x > 0 */
else
  r = -1;       /* x <= 0 */
```

El comando `if (cond) then-stmt else else-stmt` ejecuta `then-stmt` si `cond` evalúa a un valor diferente de cero (falso) o `else-stmt` en otro caso. En el comando `if` la cláusula `else else-stmt` es opcional.

La sentencia `switch` permite ejecutar diferente código por *casos*:

```
switch (x) {
  case v1:
            ...     /* stmts ejecuted when x == v1 */
          break;
  case v2:
            ...     /* stmts ejecuted when x == v2 */
          break;
  ...
  default:
            ...     /* ejecuted when x != v1, x != v2, ... */
}
```

La sentencia `break` realiza un salto al final de la sentencia que la contiene. `break` puede utilizarse en las sentencias `switch` y en las de iteración.

Si en un caso se omite `break`, se ejecutarán las sentencias del caso siguiente.

7.3. Iteración

C ofrece tres sentencias de iteración. La primera, conocida en otros lenguajes como iteración definida permite ejecutar código un número determinado de veces.

La sintaxis es el sigiente:

```
for(initialization ; condition; step) stmt;
```

Ejemplo:

```
int v[N];

for (int i=0; i<N; i++)
  v[i] = 0;
```

Cada cláusula de la sentencia `for` se separa por `;`. Cada cláusula puede contener una secuencia (operador `,`), posiblemente vacía, de expresiones. Otro ejemplo:

```
int v[N], *p, i;

for (i=0, p=v; p<v+N; p++,i++)
  *p = 0;
/* here, i == N */
```

En realizad la sentencia `for` de C es mas expresiva a sus contrapartes en otros lenguajes ya que la condición puede ser totalmente arbitraria. La condición vacía es equivalente a verdadero. Ejemplo (ciclo infinito):

```
for(;;) /* some stmt (maybe empty) */ ;
```

La sentencia `while (cond) stmt` corresponde a la iteración indefinida, es decir que puede ciclar cero o más veces. Ejemplo:

```
while (x<0)
  x += f(x);
```

La sentencia `do-while` corresponde a la iteración indefinida ejecutando su cuerpo al menos una vez. Ejemplo:

```
do {
  x += f(x);
} while (x<0);
```

En el cuerpo de las sentencias de iteración pueden aparecer sentencias de *ruptura*, `break`, la cual salta al final (fuera) del ciclo, o `continue`, la cual salta al principio del ciclo.

Los comandos `break` y `continue` actúan como *saltos* estructurados, es decir teniendo en cuenta el contexto en que se encuentran.

8. Directivas del pre-procesador

El compilador C viene acompañado de una herramienta complementaria conocida como el pre-procesador de C. Este es utilizado en el proceso de compilación para procesar las directivas que puedan encontrase en un archivo fuente.

Las directivas se distinguen en el texto porque comienzan con el caracter #.

En realidad, el pre-procesador tiene poco que ver con el lenguaje C y podría utilizarse para procesar otros tipos de archivos de texto.

Tipos de directivas:

- Inclusión de archivos
- Selección de texto (compilación) condicional
- Macros

A continuación se muestra un ejemplo conteniendo una variedad de directivas.

```
#if PLATFORM == GNU
#include <unistd.h>
#elif PLATFORM == APPLE
#include <cocoa.h>
#else
...
#endif
```

La directiva *#define* que permite definir macros (*object-like* o *function-like*).

Es posible definir un símbolo (objecto), por ejemplo para luego realizar compilación condicional:

```
#define MAXSIZE 50
```

Para el ejemplo anterior es posible pedirle al compilador que define un símbolo (en gcc lo hace la opción `-D`). Por ejemplo: `gcc -DPLATFORM=GNU -o myprog myprog.c`

Es posible definir símbolos sin un valor asociado como por ejemplo para evitar múltiples inclusiones de declaraciones como se muestra en la sección siguiente.

Una macro tipo función se muestra en el siguiente ejemplo:

```
#define max(x,y) ( (x>y)? x : y )
```

(notar la definición entre paréntesis: es recomendable usar paréntesis para independizarse del contexto de su expansión y evitar problemas de asociatividad y precedencia).

La diferencia entre una macro y una función es que una macro es expandida a su definición textualmente. Ejemplo:

```
int a,b,c;

... max(a+b,b+c) ... /* ( (a+b > b+c)? a+b : b+c ) */
```

por lo que no es una llamada a función y esa misma expresión puede usarse para diferentes tipos de datos. Las macros se expanden *en línea*[12] y a veces son utilizadas para derles nombres a patrones de código usados generalmente pero lo suficientemente compactos como para evitar una llamada a función y para definir símbolos y constantes.

A veces es difícil determinar si algo (por ejemplo, una constante) debería ser definido con una directiva o con una declaración `const`. Con respecto a uso de memoria no habrá diferencia con compiladores modernos, pero sí si es deseable definir el valor de ese símbolo en las opciones del comando de compilación.

9. Modularidad

Los programas grandes, generalmente requieren que se dividan en módulos funcionales mas pequeños para permitir la comprensión y manejo del sistema.

Un módulo es una *unidad de compliación*. Esto quiere decir que en el proceso de compilación cada módulo (archivo .c) se compilará a su correspondiente archivo objeto (.o), el cual podrá luego ser parte de un ejecutable o de una biblioteca.

Un módulo consiste en dos partes:

- **Interface**: Describe las funciones y posiblemente estructuras de datos que el módulo ofrece al resto del sistema, ocultando los detalles que no deberían ser de interés para el usuario del módulo.
- **Implementación**: Definiciones de datos y funciones.

`C` no incluye parabras reservadas para la definición de módulos como en otros lenguajes (Ada, Object Pascal, C++, ...), sino que los programadores C siguen una convención de cómo describir un módulo en sus dos componentes. La interface de un módulo se escribe en un archivo `.h`[13] y la implementación en un archivo cuyo nombre tiene como sufijo (o extensión en algunos sistemas) `.c`.

[12] Generalmente esto se denomina *sustitución textual*

[13] Por `header` o cabecera.

En un archivo de cabecera se declaran constantes simbólicas (a veces con la directiva `#define`), tipos de datos (`typedef, struct, enum`, ...) y se declaran los *perfiles* (prototipos) de las funciones. Ejemplo:

```
/* file: employee.h */
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#define N 41

struct employee {
  int id;
  char name[N];
  char address[N];
  float salary;
};

/* 'employee' is alias of 'struct employee' */
typedef struct employee employee;

void save_employee(employee * e);
employee find_employee(char * name);
...
#endif
```

El archivo de implementación (.c) tendría la siguiente estructura:

```
/* file: employee.c */

#include "dbutils.h"

#define N 41

struct employee_s {
  int id;
  char name[N];
  char address[N];
  float salary;
};

typedef struct employee_s employee;

void save_employee(employee * e)
{
        ...
        save_int_in_db("employee.id", e->id);
        save_str_in_db("employee.name", e->name);
        ...
}
```

```
employee find_employee(char * name)
{

}

static void save_int_in_db(char * field_name, int value)
{
        ...
}
...
```

A menudo los programadores C novatos cometen los siguientes errores en la definición de módulos:

- En el .h definen variables. Si el .h se incluye en varios archivos .c, algunos compiladores y/o linkers arrojan el error *múltiple definición del identificador*

  Un .h debe contener sólo *declaraciones de tipos*, no *definiciones de variables*.

- No proteger el archivo de cabecera del problema de *múltiples inclusiones* con las directivas `#ifndef` y `#define`.

- En un archivo .h incluyen un archivo .c (horror).

Un programa que incluye varios módulos (ej: `main.c` y `mymod.c`) en el cual `main.c` incluye (usa) `mymod.h` y asumiendo que usará funciones definidas en la biblioteca `math`[14], se compilará con el siguiente comando:

```
cc -o myprog.exe main.c mymod.c -lm
```

donde `myprog.exe`[15] es el archivo de salida (ejecutable producido por el compilador) y la opción `-l` (link) establece que el programa se debe enlazar con la biblioteca (pre-compilada) `m` (math).

Un programa generalmente debe hacer entrada-salida, usar las funciones de biblioteca de manipulación de strings, etc. Estas funciones están incluidas en la *biblioteca estándar*. Por eso es muy común encontrar que un programa C incluya los siguientes archivos de cabecera:

```
#include <stdio.h>  /* printf, scanf, ... */
#include <string.h> /* strcpy, strcmp, ... */
...
```

[14]Esta biblioteca define funciones como `sin(), cos(), sqrt(), ...`

[15]En MS-Windows deberá tener la extensión .exe. No es necesario en sistemas tipo UNIX (Linux, MAC-OS, ...).

El compilador C automáticamente le pide al `linker` que enlace un programa con la biblioteca estándar, excepto que se lo instruya para no hacerlo (ej: gcc -nostdlib ...).

10. Ambiente de ejecución

El compilador, con la ayuda de otras herramientas (assembler, linker, etc) genera un programa (archivo ejecutable) que contiene al menos dos partes: las instrucciones de programa, generadas a partir de las definiciones de las funciones y los valores de las variables globales inicializadas. El programa también contiene información adicional necesaria para que el sistema operativo pueda ejecutarlo (convertirlo en proceso), tal como la dirección de memoria de la primera instrucción a ejecutar (punto de entrada del programa), el tamaño del espacio de memoria requerido para almacenar todas las variables locales, etc.

La figura 10 muestra el esquema de un proceso en ejecución.

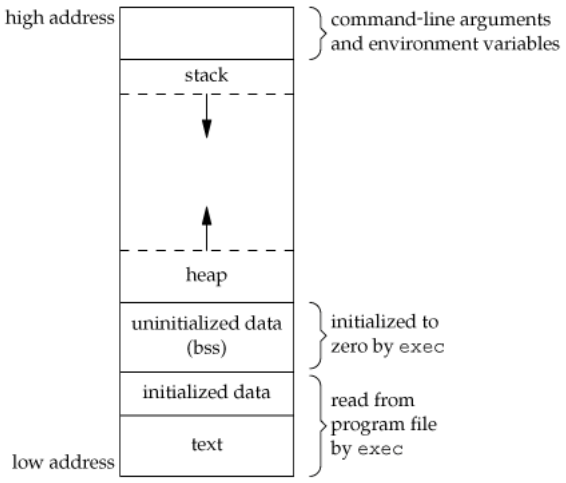

Figura 3: Representación en memoria de un proceso (UNIX).

La pila (*stack*) se utiliza para representar información de control (direcciones de retorno y comienzo del frame del invocante) necesario para manejar las llamadas/retornos a/de subrutinas y para almacenar los valores de los argumentos y variables locales automáticas de cada función.

La pila es en realidad una pila de *registros de activación*.

Cada vez que se llama a una función, se apila un nuevo registro de activación, el que desaparece luego del retorno. Esta estructura de control se arma y desarma en forma colaborativa por el código generado por el compilador en cada llamada (invocante)[16] y la rutina invocada[17].

La figura 10 muestra el esquema de un registro de activación de una función.

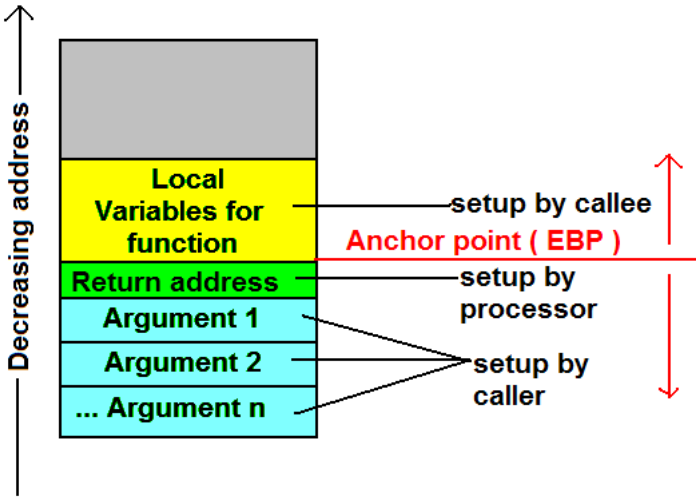

Figura 4: Stack frame (x86).

En GNU-Linux en la arquitectura x86 se asume por convención que el valor de retorno de una función queda en el registro `EAX` de la CPU.

El área de *heap* se reserva para almacenar las variables creadas dinámicamente. C no tiene incluidos mecanismos para manejo de variables heap pero la biblioteca estándar incluye funciones de utilidad (`malloc()`, `free()`) para esto, o sea que el heap es manejado enteramente por la biblioteca y el lenguaje no da (ni necesita hacerlo) ningún soporte para este mecanismo.

El siguiente ejemplo muestra un ejemplo de uso de variables heap.

```
#include <stdlib.h>
typedef struct person { int id; char name[31]; } person;
```

[16]El invocante apila los valores de los parámetros actuales, realiza la llamada (en x86, con la instrucción `call`) y luego libera el espacio de los parámetros.

[17]Cada rutina tiene un preámbulo que reserva espacio para las variables automáticas y lo libera antes del retorno.

```
person * create_person(void)
{
        person * p = malloc(sizeof(person));
        p->id = 0;
        p->name[0] = 0; /* empty string */
        return p;
}

int main(void)
{
    person * person_ptr = create_person();
    /* use person... */
    free(person_ptr);
    /* ... */
}
```

En un sistema tipo UNIX, un programa se ejecuta con la llamada al sistema `execve(prog,args,env)`, donde `prog` es el nombre del archivo ejecutable, `args` es un arreglo de strings (los argumentos) y `env` es un arreglo de strings con la forma ‘‘`variable=valor`’’ (variables del ambiente).

El sistema deja disponibles los argumentos y las variables de ambiente en un área de memoria y hace posible su acceso por medio de los argumentos de la función `main`.

En estos sistemas la función `main` tiene el perfil completo de la forma:

```
int main(int, char *[], char **);
```

donde el primer parámetro es el número de argumentos, el segundo es el arreglo de strings con los argumentos y el tercero es el arreglo de strings de variables de ambiente[18].

El siguiente programa de ejemplo muestra los argumentos y las variables de ambiente.

```
#include <stdio.h>

int main(int argc, char * argv[], char **envv)
{
        int i;

        /* print arguments */
        for (i=0; i<argc; i++)
                printf("arg %d: %s\n", i, argv[i]);

        /* print environment variables */
```

[18]Este último arreglo tiene el último puntero en cero (misma convención que para los strings).

```
        printf("\n\nEnvironment variables:\n");
        for (i=0; envv[i] != NULL; i++)
                printf("%s\n", envv[i]);

        return 0;
}
```

11. Herramientas

Cada sistema operativo define el formato de los archivos objecto (módulos complidados), bibliotecas (contenedor de módulos, generalmente con un índice) y los programas (archivos ejecutables).

Por ejemplo, en los sistemas tipo UNIX (ej: GNU-Linux) se utiliza *Executable and Linking Format*[4]. Este formato es un estándar abierto.

La figura 11 muestra esquemáticamente la estructura de un archivo ELF.

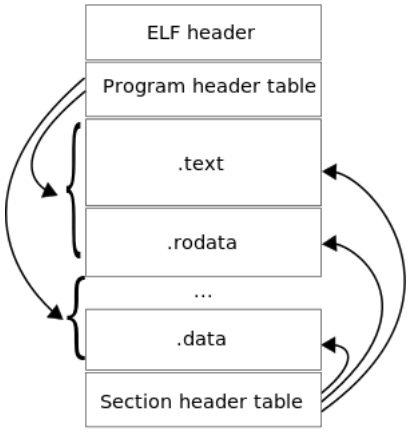

Figura 5: Esquema de un binario ELF.

Un archivo `ELF` tiene dos vistas:

- `Program header`: contiene información para la ejecución del programa (usado generalmente por la llamada al sistema `exec` del sistema operativo).

  Contiene la dirección del *punto de entrada* y los segmentos de código y datos globales (`.text, .rodata, .data`). `Exec` carga estos segmentos en la memoria desde el archivo, asigna memoria para la pila y configura el estado inicial del proceso para que cuando tome el control comience por el punto de entrada.

- `Sections header`: Contiene diferentes secciones de información y una tabla de símbolos. Cada símbolo, módulo y otras entidades tienen una descripción en una o más secciones.

  Esta información es útil para el linker para poder resolver referencias externas (ej: invocación de una función en un archivo objeto que está definida en otro) para poder fusionar archivos objetos para generar el archivo ejecutable final.

Existen varias utilidades para analizar y manipular archivos `ELF` como son `objdump`, `readelf`, `elfedit`, y `nm`.

11.1. Compilador

Generalmente el compilador es el responsable de generar el código objecto de cada módulo, produciendo archivos objetos[19]. Estos se *combinan* con código incluido en otros módulos o bibliotecas. Este proceso se llama *linking (enlazado)*.

La figura 11.1 muestra los pasos de compilación.

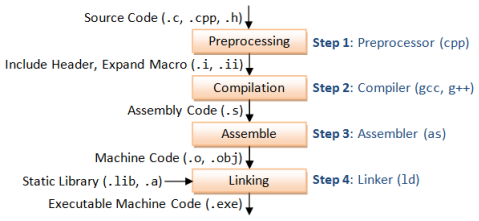

Figura 6: Proceso de compilación (GCC).

11.2. Linker

El linker es el encargado de tomar archivos objeto (.o u .obj) y bibliotecas (.a, .so, .dll), combinarlos y generar un ejecutable.

Un archivo objeto no puede ser directamente ejecutado ya que contiene *referencias externas no resueltas*, es decir, símbolos asociados a direcciones de memoria aún desconocidas, como por ejemplo, símbolos (variables, funciones, etc) definidos en otros módulos.

[19]En sistemas tipo UNIX estos archivos tienen sufijo `.o`, en MS-Windows `.obj`

La vinculación involucra la concatenación de los segmentos de código y de datos (globales) de cada archivo objeto, la asignación de direcciones de memoria de los símbolos externos (exportados por cada módulo) y construcción de una tabla de símbolos externos.

Generalmente es invocado automáticamente por el comando de compilación pero es una aplicación autónoma[20].

Cuando un archivo objeto se enlaza estáticamente, éste se *incluye* con las referencias externas resueltas en el ejecutable producido.

El linker puede tomar archivos objetos desde una biblioteca, la cual es un contenedor de módulos (archivos objeto) y pueden ser de dos tipos:

- **Estática** (archivos `.a` o `.lib`): Se incluye su contenido (código y datos) en el executable.

  Ventaja: sólo hace falta redistribuir el ejecutable final.

  Desventaja: las bibliotecas de uso común (ej: libc) estarían incluidas en cada programa, por que habría muchísima redundancia en los contenidos y código repetido durante la ejecución de procesos.

- **Dinámica** (archivos `.so` o `.dll`): Su contenido no se incluye en el programa producido. El programa sólo incluye una sección con información necesaria sobre las dependencias (bibliotecas) requeridas y una tabla de referencias externas (ej: identificadores de funciones) que deberán resolverse dinámicamente (en tiempo de ejecución) por el sistema operativo cuando el programe se carge en memoria.

  Ventajas: programas mas pequeños. Se mantiene una sola copia en memoria de la biblioteca. La biblioteca se puede cargar bajo demanda (ej: en la primera invocación a una función)y por partes.

  Desventajas: hay que distribuir el programa + las bibliotecas. Es necesaria una maquinaria mas compleja en el sistema operativo (memoria virtual, incluir un linker, etc). Hay una pequeña penalización en el tiempo de carga de un programa (linking).

Muchas bibliotecas se compilan en ambas versiones, estática y dinámica. En ese caso, generalmente el linker selecciona la biblioteca dinámica, excepto si se le indica explícitamente.

[20]En sistemas GNU, el linker es el comando `ld`

Cuando se compila un programa con varios módulos (archivos fuente `.c`), se genera un archivo `.o` por cada módulo y si el compilador invocará al linker para generar el ejecutable a partir de los archivos objeto.

Cuando se está construyendo una biblioteca ningún módulo incluirá la función `main()`, por lo que debe instruirse al compilador que no invoque al linker[21].

Una biblioteca estática `mytools` en un sistema tipo UNIX (ej: Linux) con `gcc` se construye de la siguiente forma:

```
gcc -c f1.c f2.c
ar rcs libmytools.a f1.o f2.o
```

El comando `ar` (*archivador*) genera un *contenedor* de otros archivos y puede generar índices a archivos y tablas de símbolos definidos en archivos objetos.

No confundirlo con el comando `tar` el cual es similar pero éste último no está diseñado para ser usado por un linker sino como un archivador mas simple y general, como por ejemplo, apara distribuir aplicaciones o hacer copias de respaldo (backups).

Una biblioteca compartida (dinámica) `mylibtools` se genera con el comando:

```
gcc -shared -o libmytools.so f1.o f2.o
```

Se debe notar que en este último comando sólo invoca al linker (*ld*).

En los sistemas tipo UNIX el nombre de una biblioteca debe tener el prefijo `lib`.

11.3. Debuggers y profilers

Un debugger (depurador) permite ejecutar un programa hasta un *breakpoint*. De ahí en mas permite que el usuario ejecute el proceso paso a paso o hasta alcanzar otro breakpoint.

Durante uns sesión de debugging es posible visualizar y hasta cambiar valores de variables, inspeccionar el stack, etc.

El objetivo del debugging de programas es generalmente tratar de encontrar un error de difícil detección por simple análisis del programa fuente.

En un sistema GNU el debugger es el comando `gdb`. Para que un programa pueda ser depurado, el compilador deberá incluir información adicional en el código generado necesaria para la depuración. En el `gcc` esto se logra

[21] En `gcc` es la opción `-c`.

añadiendo la opción `-g`.

El programa `gdb` es interactivo y permite cargar un programa, definir breakpoints, correrlo hasta un determinado punto de programa, inspeccionar la memoria usando identificadores de variables, etc. También es posible inspeccionar cuál fue el comportamiento (reproducir su ejecución) de un proceso que fue terminado por el sistema operativo por algún error irrecuperable (crash). En esos casos los sistemas operativos tipo UNIX generan una *imagen de la memoria* (`.core`) del proceso al momento del error para que luego pueda analizarse con el debugger.

En particular `gdb` se invoca de la forma:

```
gdb program
gdb program core-file
gdb -p pid
```

El último comando permite depurar un proceso ya en ejecución.
La tabla 11.3 describe algunos comandos del `gdb`.

Comando	**Descripción**
break file:function	Define un breakpoint en la función dada
run args	Inicia la ejecución
bt	(Backtrack) muestra el stack
print expr	Muestra el valor de la expresión
c	Continúa la ejecución
next	Ejecuta la próxima línea de programa
step	Ejecuta un paso (si es llamada a función, entra en ésta)
quit	Sale del gdb

Figura 7: Algunos comandos del `gdb`.

En GNU-Linux el programa `Data Display Debugger (ddd)` es un front-end gráfico muy fácil de utilizar.

Otras herramientas útiles para analizar el rendimiento de programas son los *perfiladores (profilers)*, como por ejemplo `gprof`, el profiler de GNU.

Cuando un programa se compila con soporte de profiling (opción `-pg` del `gcc`), cuando se ejecuta se genera información en cada llamada a función. La información recolectada durante la ejecución se almacena en el archivo `gmon.out` y se puede analizar con `gprof`, el cual da información sobre el per-

fil de rendimiento general (tiempos de ejecución en cada función, cantidad de llamadas de cada función, etc.) y además genera el grafo de llamadas (*call graph*) en forma textual. Es posible generar un gráfico usando `gprof2dot`.

Otra herramienta de perfilado ampliamente usado es `gcov` el cual genera información sobre la cobertura de código y frecuencias de ejecución.

11.4. Make

En los proyectos de software grandes, con muchos archivos fuentes, el tiempo de compilación de todo el sistema puede ser muy elevado. El programa `make` toma un archivo (por omisión `Makefile`), el cual contiene información de cómo debería construirse el sistema a partir de los archivos fuentes.

En realidad un *Makefile* permite definir reglas de construcción de objetos (*targets*) definiendo sus dependencias y los comandos de construcción.

Cuando alguna de las dependencias (ej: un archivo fuente) cambia[22], el comando detecta que el objeto destino correspondiente debe reconstruirse invocando los comandos especificados en la regla.

Las reglas pueden ser dependendientes en forma transitiva por lo que forman un *grafo de dependencias*.

El programa `make` permite reconstruir sólo las partes requeridas, permitiendo la recompilación sólo de los archivos modificados.

Una regla tiene la siguiente forma:

```
target: dependencies
<tab>commands
```

Ejemplo de `Makefile`:

```
# build the application linking with mylib
myapp: main.c libmylib.so
        gcc -o myapp -Wall main.c -L/home/myuser/dev -lmylib

# build object files
p1.o: p1.h p1.c
        gcc -c p1.c

p2.o: p2.h p2.c
        gcc -c p2.c
```

[22]La detección se realiza comparando las fechas y horas de los objetivos y las dependencias.

```
# build the static and shared libraries
libmylib.a: p1.o p2.o
        ar rcs libmylib.a p1.o p2.o

libmylib.so: p1.o p2.o
        gcc -shared -o libmylib.so p1.o p2.o
```

El comando `make myapp` reconstruye el objetivo dado (si se omite el argumento, reconstruirá el primer objetivo del `Makefile`).

11.5. GNU Binutils y autotools

El paquete `binutils`[?] contiene un conjunto de utilidades para la manipulación de archivos binarios como `ar`, `dlltool`, `gprof`, `nm`, `objdump`, `readelf`, `windres`, ...

El paquete conocido como `autotools`[?] es el GNU Build System el cual es un conjunto de herramientas para crear y compilar paquetes de software de código fuente portable a diferentes sistemas operativos, en particular aquellos tipo UNIX (aunque también existen para MS-Windows).

Este paquete contiene dos herramientas principales:

- `Autoconf`: Toma como entrada archivos `configure.ac` (o `configure.in`) y `Makefile.in`, los cuales contiene macros que generalmente chequean las dependencias requeridas y genera un archivo de script del shell bash `.configure`.

  Al correr el script `configure`, éste genera el archivo `Makefile` que será usado por el comando `make`.

- `Automake`: Permite crear un archivo `Makefile.in` desde un archivo de macros `Makefile.am`.

  En un archivo `Makefile.am` generalmente se especifican los nombres de los ejecutables a producir, la lista de los archivos fuente, las opciones al compilador y al linker, entre otras cosas.

Estas herramientas permiten que un paquete se distribuya generalmente en formato `.tgz` y debería poder ser instalado mediante los siguientes pasos:

```
./configure
make
make install
```

Referencias

[1] Brian Kernighan, Dennis Ritchie. *El Lenguaje de Programación C*. Pearson Education. 1998. ISBN: 0133086216, 9780133086218.

[2] ISO/IEC 9899. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`

[3] ISO/IEC 9899. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`

[4] Executable and Linking Format. `http://refspecs.linuxbase.org/elf/elf.pdf`

[5] GNU Binutils. `http://www.gnu.org/software/binutils/`

[6] GNU Build System (autotools). `http://www.gnu.org/savannah-checkouts/gnu/automake/manual/html_node/Autotools-Introduction.html`