De Pascal a C

Gonzalo Soriano
gonzalo_soriano@yahoo.com.ar

13 de diciembre de 2007

Índice

1. Introducción

El lenguaje de programación **C** fue creado en 1969 por Dennis Ritchie [1] y Ken Thompson [2] en los Laboratorios Bell como evolución de un lenguaje anterior llamado B [3]. **C** es un lenguaje de programación estructural al estilo de Pascal [4], pero con una sintaxis diferente. Puede ser catalogado de alto o bajo nivel, dependiendo de cómo se lo use.
En este apunte se tratará de respetar el estándar ANSI C, por lo que cuando se usen funciones o cosas que no pertenezcan a ese estándar se comentaran.

2. Diferencias entre C y Pascal

- La primer diferencia entre C y Pascal es que C es case sensitive; es decir, sensible al caso. Por este motivo, una variable llamada *entero* es distinta a *Entero*; y una función llamada *factorial* será distinta de otra llamada *Factorial*.
- Pascal y C son dos lenguajes estructurales, pero tienen sintaxis distintas. Un claro ejemplo se ve cuando uno busca el bloque principal del programa. En Pascal el bloque principal se encuentra entre las sentencias *Begin* y *End.*, pero en C el bloque principal es una función llamada *main*. El main retorna un entero y recibe dos parámetros llamados *argc* y *argv*, que son de tipo entero y char** respectivamente. El tipo char** también se puede encontrar como char* [ ].
  Argc será el número de parámetros que van en la línea de comandos (cada parámetro separado por espacios), y argv será una matriz de caracteres que contendrá los caracteres de esos parámetros.
- Otra gran diferencia entre estos lenguajes, es la forma de delimitar el código, es decir, cuando en Pascal queríamos que un if ejecute varias sentencias las encerrábamos entre un *begin ... end*; en C para eso usaremos {...}.
- Y como las {} ahora se usan para encerrar varias sentencias, ya no se pueden usar para hacer comentarios, por lo que ahora se usa `/*` para abrir un comentario y `*/` para cerrarlo, y en el caso de compilar el código con un compilador de C++, se puede usar `//` para comentar toda una línea.

2.1. Palabras reservadas en C

Como todos los lenguajes de programación, C utiliza ciertos nombres para designar funciones, dichos nombres se conocen como palabras reservadas; lo que significa que una variable no puede llamarse con uno de estos nombres.

auto	const	else	goto	return	struct	void
asm	continue	enum	if	short	switch	volatile
break	default	extern	int	signed	typedef	while
case	do	float	long	sizeof	union	
char	double	for	register	static	unsigned	

Cuadro 1: Palabras reservadas en C.

2.2. Definición de Variables

Antes de ver cómo declarar las variables, tenemos que saber que tipos podemos usar; para eso primero veamos una tabla que relaciona los tipos de Pascal con los de C.

Pascal	C
integer	int
byte	short int
longint	long int
word	unsigned int
real	float
double	double
extented	long double
char	char
Pchar	char*
string	—
boolean	—
—	void
Pointer	void*

Cuadro 2: Equivalencias entre tipos.

Cabe mencionar que el tipo char puede considerarse como un entero con rango [-128, 127]. También es importante mencionar que una variable puede inicializarse en su declaración; y que si bien no existe el tipo boolean en C, se usan los enteros con ese fin. Si un entero vale 0 equivale al *false* de los boolean; y si es distinto de 0, sería *true*.
Ahora que ya sabemos cómo son los tipos en C, podemos ver cómo se declaran las variables. A diferencia de Pascal, en C primero se le declara el tipo, se deja un espacio, y después el nombre. Si queremos declarar una variable de tipo entero, la sentencia será:

```
int entero;
```

Y a diferencia de Pascal, C no tiene una parte identificada para la declaración de variables; se deben declarar en el bloque de sentencias, pero siempre antes de la primer instrucción o sentencia. Al igual que en Pascal, el alcance de las variables está delimitado por el bloque al que pertenecen.

2.3. Funciones de Entrada/Salida

Así como en Pascal están el *write* y el *read*, C tiene sus propias funciones de entrada/salida; y son: *printf* y *scanf*; aunque también existen *getchar*, *putchar*, *gets* y *puts*. Para hacer uso de cualquiera de las funciones recién mencionadas, hay que incluir una librería (el equivalente a las Unit de Pascal) llamada *stdio.h* y se incluye poniendo `#include <stdio.h>` al principio.
La función **printf** toma como argumentos una cadena de caracteres, y opcionalmente una lista de variables a ser impresas. Las variables se imprimen de acuerdo a donde se especificó en la cadena. El formato general es:

```
int printf("cadena_de_caracteres", lista de variables);
```

donde cadena_de_caracteres puede contener una especificación de conversión de formato (a la que precede con el símbolo %) y caracteres normales.
La función **scanf** es para entrada de propósito general con formato. El formato

Especificador	**Tipo:**
`%c`	Carácter ASCII correspondiente.
`%s`	Cadena de caracteres terminada en el carácter nulo (`\0`).
`\n`	Carácter fin de línea.
`\r`	Carácter retorno del carro.
`\t`	Carácter tabulador horizontal.
`\v`	Carácter tabulador vertical.
`\a`	Timbre.
`\o`	Carácter nulo.
`\\`	Barra invertida.
`\"`	Comillas.
`%d %i`	Decimal con signo de un entero.
`%u`	Decimal sin signo de un entero.
`%ld`	Entero largo con signo.
`%lu`	Entero largo sin signo.
`%f`	Float o double.
`%e`	Float o double, con notación científica.
`%g`	Coma flotante, usando la notación que requiera menor espacio.
`%x`	Hexadecimal sin signo.

Cuadro 3: Especificaciones de conversión.

general es:

```
int scanf("cadena_de_caracteres", lista de punteros a variables);
```

La función lee de la entrada estándar elementos que son convertidos, según las especificaciones de `cadena_de_caracteres`, y asignados a las variables cuyas direcciones se pasan en la lista.

Ejemplo 1:
Ahora veamos nuestro primer programa en C:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hola mundo, hoy es %d/%d/%d\n",29,06,2007);
}
```

Ejemplo 2:
Hacer un programa que lea un número por entrada estándar y lo vuelva a imprimir.

```
#include <stdio.h>

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

    printf("Ingrese un número: ");
    scanf("%d", &entero);
    printf("El número ingresado fue el %d.\n", entero);
}
```

2.4. Estructuras de selección

2.4.1. if else

```
if (condición)
   sentencia1; /* Ver que termina con ; */
else
   sentencia2;
```

Al igual que en Pascal, el if puede presentarse sin el else:

```
if (condición)
   sentencia;
```

2.4.2. case

```
switch (variable)
{
   case valor1: sentencia1; break;
   case valor2: sentencia2; break;
   case valor3: sentencia3; break;
   case valor4: sentencia4; break;
   default: sentencia5; break;
}
```

2.5. Ciclos repetitivos

2.5.1. for

```
for (condición inicial; condición de corte; modificación de la variable)
   sentencia;
```

2.5.2. while

```
while (condición)
   sentencia;
```

2.5.3. repeat

```
do
   sentencia;
while (condición);
```

2.6. Operadores en C

Operador en Pascal	Operador en C	Nombre
NOT	- ~ !	Negación / Complemento.
sizeOf	sizeof	Obtiene el tamaño en bytes.
^	*	Indirección.
@	&	Dirección.
+ -	+ -	Suma y resta.
*	*	Multiplicación.
/	/	División.
div	/	División Entera.
mod	%	Resto.
inc(var)	var++	Suma unitaria.
dec(var)	var--	Resta unitaria.
:=	=	Asignación.
< > <= >=	< > <= >=	Operadores de relación.
=	==	Comparación por igual.
<>	!=	Comparación con distinto.
AND OR	&& ||	AND y OR lógicos.

Cuadro 4: Operadores en C.

Ejemplo 3:
Un ejemplo del if.

```
#include <stdio.h>

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

    printf("Ingrese un número: ");
    scanf("%d", &entero);
    if (entero) /* Uso un entero como boolean. */
       printf("El número que ingresaste es %d, y es distinto de 0.\n", entero);
    else
       printf("El número que ingresaste es 0.\n");
}
```

Ejemplo 4:
Un ejemplo del case.

```
#include <stdio.h>

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

    printf("Ingrese un número del 1 al 5: ");
```

```
    scanf("%d", &entero);
    switch (entero)
    {
        case 1: printf("Ingresaste el 1.\n"); break;
        case 2: printf("Ingresaste el 2.\n"); break;
        case 3: printf("Ingresaste el 3.\n"); break;
        case 4: printf("Ingresaste el 4.\n"); break;
        case 5: printf("Ingresaste el 5.\n"); break;
        default: printf("Ingresaste cualquiera.\n"); break;

    }
}
```

Ejemplo 5:
Un ejemplo del for.

```
#include <stdio.h>

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

    printf("Ingrese un número: ");
    scanf("%d", &entero);
    for(i=0; i<entero; i++)
    {
        printf("Imprimiendo el número %d y me faltan %d.\n", i, entero-i-1);
    }
}
```

Ejemplo 6:
Un ejemplo del while.

```
#include <stdio.h>

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

    printf("Ingrese un número: ");
    scanf("%d", &entero);
    while(entero != 5)
    {
        printf("Ingrese un número (corta con 5):");
        scanf("%d", &entero);
    }
}
```

Ejemplo 7:
Un ejemplo del do-while.

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])
{
    int entero;

    do
    {
        printf("Ingrese un número (corta con 5):");
        scanf("%d", &entero);
    } while(entero != 5);
}
```

2.7. Estructuras de datos

2.7.1. Registros

En Pascal cuando queríamos agrupar datos de distintos tipos usábamos los registros, análogamente en C usaremos los struct. Para eso veamos primero su declaración:

```
struct nombre_estructura
{
   declaración de variables;
};
```

Por ejemplo, si quisiéramos declarar un struct que represente los datos de un alumno, haríamos:

```
struct tAlumno
{
   unsigned int padron;
   unsigned long int dni;
   char *nombre;
};
```

Pero eso es solo para la declaración del tipo, para declarar una variable del tipo tAlumno tenemos que hacer:

```
struct tAlumno alumno1;
```

Para acceder a un campo de la estructura, debemos hacer igual que en Pascal; simplemente con un punto y el nombre del campo:

```
    struct tAlumno alumno1;
    int p;

    alumno1.padron = 0;
    p = alumno1.padron;
```

Pero puede ser muy molesto tener que poner siempre `struct` antes del tipo que se quiere usar, para lo que usan los `typedef`.

2.7.2. Typedef

Al igual que en Pascal, en C también podemos definir nuestros propios tipos usando la palabra reservada `typedef`:

```
typedef definción_del_tipo nombre_del_tipo;
```

Por ejemplo, si queremos seguir usando la palabra *byte* para los enteros de rango -128 a 127; podemos hacer el siguiente *typedef*:

```
typedef short int byte;
```

Y para el caso de los struct; haríamos:

```
typedef struct
{
   unsigned int padron;
   unsigned long int dni;
   char *nombre;
} tAlumno;
```

ó

```
struct tAlumno
{
   unsigned int padron;
   unsigned long int dni;
   char *nombre;
};
typedef struct tAlumnos TAlumno;
```

3. Funciones y Procedimientos

En C no existen procedimientos, son todas funciones. Los procedimientos son funciones que retornan void. Y la declaración de una función es:

```
tipo_retorno nombreFuncion (tipo1 x1, tipo1 x2, ..., tipoN xN);
```

Pero lo que declaramos recién es como cuando en Pascal declarábamos una función y le poníamos el forward para decirle que la íbamos a definir más abajo. La definición sería:

```
tipo_retorno nombreFuncion (tipo1 x1, tipo1 x2, ..., tipoN xN)
{
   /* Variables. */

   /* Sentencias. */

   return valor_retorno; /* Si tipo_retorno es void, esto no va. */
}
```

Ejemplo 8:
Hacer la función factorial.

```
#include <stdio.h>

int factorial(int n)
{
    int i, fac=1;

    for(i=1; i < n+1; i++)
        fac *= i;

    return fac;
}

int main(int argc, char *argv[])
{
    int entero, resultado;

    printf("Ingrese un número: ");
    scanf("%d", &entero);
    resultado = factorial(entero);
    printf("El factorial de %d es %d.\n", entero, resultado);
}
```

Ejemplo 9:
Hacer un procedimiento que imprima el factorial.

```
#include <stdio.h>

void factorial(int n)
{
    int i, fac=1;

    for(i=1; i < n+1; i++)
        fac *= i;
    printf("El factorial de %d es %d.\n", n, fac);
}

int main(int argc, char *argv[])
{
    int entero, resultado;

    printf("Ingrese un número: ");
    scanf("%d", &entero);
    factorial(entero);
}
```

C no posee pasaje de parámetros por referencia, por lo que todos son pasajes por valor. Lo que se hace cuando se quiere hacer un pasaje por referencia es usar un puntero que apunte a la posición de memoria asignada a la variable.

Ejemplo 10:

```
void intercambio (int* x, int* y)
{
   int temp = *x;
   *x = *y;
   *y = temp;
}
```

Y la llamada al procedimiento sería:

```
   int a, b;
   a = 10;
   b = 5;
   intercambio(&a, &b);
```

4. Vectores, Matrices y Strings

4.1. Vectores

Los vectores pueden ser de cualquier tipo de variable y su definición es:

```
tipo_variable nombre_variable[tamaño];
```

A diferencia de Pascal, en C el índice empieza en 0, no en 1.
Si no se le indica un tamaño estaremos usando memoria dinámica, por lo que tendríamos que pedírsela al Sistema Operativo. Y para acceder a una posición del vector, solo tenemos que hacer:

```
nombre_variable[posición_deseada];
```

Como el resto de las variables, un vector también se puede inicializar en su declaración:

```
tipo_variable nombre_variable[tamaño] = {valor1, ... valorN};
```

Ejemplo 11: Ejemplo de vectores.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int  i, vector[3] = {1, 2, 3};

    for(i=0; i<3; i++)
    {
        printf("Ingrese un número: ");
        scanf("%d", &vector[i]);
    }
    printf("Vector: ( ");
    for(i=0; i<3; i++)
        printf("%d ", vector[i]);
    printf(")\n");
}
```

4.2. Matrices

Las matrices son vectores, pero con más dimensiones.

```
tipo_variable nombre_variable[numero_fils][numero_cols];
```

Si queremos acceder a una posición de la matriz, solo tenemos que hacer:

```
nombre_variable[fila][columna];
```

Pero también podemos acceder a una fila entera, y lo haríamos así:

```
nombre_variable[fila];
```

Al igual que los vectores, las matrices también se puede inicializar en su declaración:

```
tipo nombre[i][j] = {{val11, ... val1N},...,{valM1, ... valMN}};
```

4.3. Strings

Los strings no existen en C, pero en lugar se utilizan las cadenas de caracteres terminadas en el carácter nulo(`\0`).
Existen varias formas de declararlas, algunas son:

```
char string1[4]; /* String de 3 caracteres, más uno del \0. */
char string2[] = "Hola mundo."; /* Creo e inicializo una cadena. */
char string3[] = {'H','o','l','a',' ','m','u','n','d','o','.','\o'}
   /* Las cadenas 2 y 3 son exactamente iguales. */
char *string4 = "cadena 4";
```

4.3.1. Algunas funciones para trabajar con los char*

Para trabajar con los string tenemos que incluir: `#include <string.h>`.

- **size_t strlen(const char *s):** Retorna el tamaño de la cadena, excluyendo el carácter nulo.
- **char *strcpy(char *destino, const char *origen):** Copia la cadena origen a la cadena destino. Si usamos el operador = copiaríamos los punteros.
- **char *strcpy(char *destino, const char *origen, size_t n):** Copia n primeros caracteres de la cadena origen a la cadena destino.
- **strcmp(const char *cad1, const char *cad2):** Compara ambas cadenas devolviendo un valor negativo si cad1 < cad2, 0 si cad1 == cad2, y un valor positivo si cad1 > cad2.
- **strcmp(const char *cad1, const char *cad2, size_t n):** Compara los primeros n caracteres de ambas cadenas devolviendo un valor negativo si cad1 < cad2, 0 si cad1 == cad2, y un valor positivo si cad1 > cad2.
- **char *strcat(char *destino, const char *origen):** Concatena la cadena origen al final de la cadena destino.

- **char *strcat(char *destino, const char *origen, size_t n):** Concatena los n primeros caracteres de la cadena origen al final de la cadena destino.
- **char *strchr(const char *cadena, int c):** Busca el primer carácter igual a **c** en la cadena.
- **char *strrchr(const char *cadena, int c):** Busca el último carácter igual a **c** en la cadena.
- **size_t strspn(const char *cadena, const char *origen):** Retorna el número de caracteres iniciales encontrados en cadena que coinciden con los de origen.
- **size_t strcspn(const char *cadena, const char *origen):** Retorna el número de caracteres iniciales encontrados en cadena que no coinciden con los de origen.
- **char *strpbrk(const char *s, const char *accept):** Retorna un puntero a la primera ocurrencia de la cadena de cualquier carácter de la cadena accept.
- **char *strstr(const char *origen, const char *buscada):** Retorna un puntero a la primera ocurrencia de la cadena buscada en origen.
- **void *memmove(void *destino, const void *origen, size_t n):** Esta ya no es exclusivo para strings, pero se puede usar para ellos. Mueve los primeros n **bytes** desde la zona apuntada por origen a la zona apuntada por destino.
- **void *memchr(const void *origen, int c, size_t n):** Busca en los primeros n bytes de origen el carácter c.
- **int memcmp(const void *s1, const void *s2, size_t n):** Compara los primeros n bytes de S1 y S2, y retorna en forma similar a strcmp.
- **void *memcpy(void *destino, const void *origen, size_t n):** Copia una los primeros n bytes desde origen a destino.
- **void *memset(void *origen, int c, size_t n):** Para rellenar n bytes a partir de origen con el carácter c.

Y ahora también tenemos dos funciones en la stdio.h, recuerden que para incluirla hacemos: `#include <stdio.h>`.

- **int sprintf(char *buffer, ".cadena_de_caracteres", lista_de_variables):** Funciona igual que **printf**, pero en lugar de escribir en la salide estándar, escribe en el buffer.
- **int sscanf(const char *buffer, ".cadena_de_caracteres", lista_de_punteros_a_variables):** Funciona igual que **scanf**, pero en lugar de leer desde la entrada estándar, lee desde el buffer.

5. Compiladores y Entornos de Desarrollo

A continuación les nombraré algunas herramientas que me parecen importantes para desarrollar, compilar y depurar código en C/C++. Trataré de comentar algo de cada herramienta en un par de renglones, pero como no use todas las herramientas que nombro o simplemente no se me ocurría que escribir, algunos comentarios son sacados de internet.

5.1. Windows

- **Dev-C++** [5] Bloodshed Dev-C++ es un entorno de desarrollo integrado (IDE por sus siglas en inglés) para programar en lenguaje C/C++. Usa MinGW que es una versión de GCC (GNU Compiler Colletion) como su compilador. Dev-C++ puede además ser usado en combinación con Cygwin y cualquier compilador basado en GCC.

- **Turbo C** [6] IDE y compilador para programar en lenguaje C desarrollado por Borland. Su primera versión es de 1987, a la que siguieron las versiones 1.5 y 2.0 de1989.
  Fue el compilador más popular para desarrollar en C en entornos MS-DOS. Se le considera el primer IDE para C disponible para dicha plataforma. Sustituido por el Turbo C++ en 1990.
  Tanto el Turbo C 2.0 como el Turbo C++ 1.0 pueden conseguirse gratuitamente en la web de Borland desde el año 2000.

- **Visual C++**.

5.2. Linux

5.2.1. Compilador

- **gcc** [7] GNU Compiler Collection es un conjunto de compiladores creados por el proyecto GNU. GCC es software libre y lo distribuye la FSF bajo la licencia GPL. Estos compiladores se consideran estándar para los sistemas operativos derivados de UNIX, de código abierto o también de propietarios, como Mac OS X. GCC requiere el conjunto de aplicaciones conocido como binutils para realizar tareas como identificar archivos objeto u obtener su tamaño para copiarlos, traducirlos o crear listas, enlazarlos, o quitarles símbolos innecesarios.
  Originalmente GCC significaba GNU C Compiler (compilador GNU para C), porque sólo compilaba el lenguaje C. Posteriormente se extendió para compilar C++, Fortran, Ada y otros.
  Se compila poniendo en línea de comandos: `gcc nombre_archivo.c -o nombre_programa`

5.2.2. Debugger

- **GDB** [8] GDB es el debugger (depurador) de GNU. Es para línea de comandos, en modo texto, tiene soporte para varios lenguajes y es muy potente.

- **DDD** [9] Data Display Debugger, o DDD, es una popular interfaz de usuario gráfica para depuradores en línea de comandos como GDB, DBX,

JDB, WDB, XDB, el depurador de Perl o el depurador de Python. DDD se rige bajo licencia GNU GPL y es software libre y tiene una GUI que permite visualizar el código fuente y una interfaz gráfica interactiva para visualizar los datos, donde las estructuras de datos son visualizadas como gráficos.

5.2.3. IDEs

Las cuatro propuestas a continuación son editores de gran potencia y versatilidad tanto para pequeños programas como para aplicaciones avanzadas. Incluyen un gestores de proyectos, asistentes de aplicaciones, depuradores interactivos y un potente editor con navegación de código y resaltado en colores.

- **KDevelop** [10] El Proyecto KDevelop surgió en 1998 con el fin de desarrollar un IDE (Entorno de desarrollo integrado) fácil de usar para KDE. Desde entonces, el IDE KDevelop está públicamente disponible bajo la GPL y soporta muchos lenguajes de programación.

- **Anjuta** [11] Anjuta es un entorno de desarrollo, o IDE, gratuito y con licencia GPL para lenguajes C/C++.

- **Eclipse** (con plugin para C++) [12] Eclipse es una plataforma de software de Código abierto independiente de una plataforma para desarrollar código Java; pero se le pude añadir un plugin para que soporte programas en C/C++.

- **NetBeans** (con plugin para C++) [13] NetBeans es una aplicación de código abierto (open source) diseñada para el desarrollo de aplicaciones fácilmente portables entre las distintas plataformas, haciendo uso de la tecnología Java. El NetBeans C/C++ Pack soporta proyectos de C/C++.

Referencias

[1] http://es.wikipedia.org/wiki/Dennis_M._Ritchie

[2] http://es.wikipedia.org/wiki/Ken_Thompson

[3] http://es.wikipedia.org/wiki/Lenguaje_de_programaci %C3 %B3n_B

[4] http://es.wikipedia.org/wiki/Lenguaje_de_programaci %C3 %B3n_Pascal

[5] http://www.bloodshed.net/dev/devcpp.html

[6] http://dn.codegear.com/article/20841

[7] http://gcc.gnu.org/

[8] http://sourceware.org/gdb/

[9] http://www.gnu.org/software/ddd/

[10] http://www.kdevelop.org/

[11] http://www.anjuta.org/

[12] http://www.eclipse.org/

[13] http://www.netbeans.org/products/cplusplus/