# 3.1.7 Definición de Recursividad

Se dice que un **proceso** es recursivo si forma parte de sí mismo ([1], pág. 465) o sea que se define en función de sí mismo. La recursión aparece en la vida diaria, en problemas matemáticos, en estructuras de datos y en muchos otros problemas.

# 3.1.8 Función recursiva

En términos de lenguaje de programación, una función es recursiva cuando se llama a sí misma ([1], pág. 466).

Un ejemplo es la función de Ackerman %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
A
\]
\end{document}, la cual está definida para todos los valores enteros no negativos %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
m
\]
\end{document} y %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
n
\]
\end{document} de la forma siguiente:

%FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\begin{eqnarray}
A(0,n)&=&n+1\nonumber\\
A(m,0)&=&A(m-1,1)\mbox{\rm\ \ \ \ \ }(m>0)\nonumber\\
A(m,n)&=&A(m-1,A(m,n-1))\mbox{\rm\ \ }(m,n>0)\nonumber
\end{eqnarray}
\end{document}

A continuación, presentamos el código correspondiente a este problema.

%FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\begin{verbatim}
/** Funci\'on Ackerman recursiva
 */
#include <stdio.h>
int Ackerman(int,int);

////////////////// main ///////////////////////
int
main(){
  int m,n,a;
  printf("C\\'alculo de A(m,n)=A(m-1,A(m,n-1))\n\n");
  printf("Valores de m y n: ");
  scanf("%d %d",&m,&n);
  a=Ackerman(m,n);
  printf("\n\nA(%d,%d) = %d\n",m,n,a);
  return 0;
}
\end{verbatim}
\end{document}

%FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\begin{verbatim}
///////////////// Ackerman ////////////////
int Ackerman(int m,int n){
  if(m==0)
    return n+1;
  else if(n==0)
    return Ackerman(m-1,1);
  else
    return Ackerman(m-1,Ackerman(m,n-1));
}
\end{verbatim}
\end{document}

El lenguaje C contiene la palabra clave **auto** ([2], pág. 19), que puede ser usada para declarar variables locales. Sin embargo, como todas las variables que no son globales son, por defecto asumidas como **auto**, esta palabra clave casi nunca se usa.

# Clases de Almacenamiento de una variable

Por defecto todas las variables llevan asociada una clase de almacenamiento ([3], pág. 83) que determina su accesibilidad y existencia. Los conceptos de accesibilidad y de existencia de las variables pueden alterase por los calificadores/especificadores:

**auto** almacenamiento automático

**register** almacenamiento en un registro

**static** almacenamiento estático

**extern** almacenamiento externo

Estos especificadores ([2], pág. 27) indican al compilador como debe almacenar la variable que le sigue. La forma general de declaración de una variable que use alguno d estos es la siguiente:

%FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\begin{verbatim}
especificador_de_almacenamiento tipo nombre_de_variable;
\end{verbatim}
\end{document}

Antes de examinar **extern,** procede una breve descripción del enlace en C. C define tres categorías de enlace: externo, interno y nulo. En general, las funciones y las variables globales se ven afectadas por un enlace externo. Esto significa que están disponibles para todos los archivos que constituyen el programa. Los objetos con ámbito de archivo declarados como **static** tienen un enlace interno. Solo son conocidos dentro del archivo en el que están declarados. Las variables locales no se ven afectadas por el enlace y por tanto son conocidas solo dentro de su propio bloque.

El uso principal de **extern** es el de especificar que un objeto está declarado con enlace externo en alguna otra parte del programa. Para entender la importancia de esto se hace necesario comprender la diferencia entre una declaración y una definición. Una declaración declara el nombre y el tipo de un objeto. Una definición hace que se disponga almacenamiento para el objeto. Un mismo objeto puede tener múltiples declaraciones, pero solo puede tener una definición.

En la mayoría de los casos, las declaraciones de las variables también son definiciones. Sin embargo, precediendo el nombre de una variable por el especificador **extern**, se puede declarar una variable sin definirla. Así, cuando se necesita referirse a una variable que ha sido definida en alguna otra parte del programa (posiblemente en otro archivo), se debe declarar la variable utilizando **extern**.

Los calificadores **auto** y **register** ([3], pág. 83)pueden ser utilizados solamente con variables locales; el calificador **extern** puede ser utilizado solo con variables globales; y el calificador **static** puede ser utilizado con variables locales y globales.

## Calificación de variables globales

Una variable declarada a nivel global ([3], pág. 83) es una definición de la variable o una referencia a una variable definida en otra parte. Las variables declaradas a nivel global son iniciadas a cero por omisión.

Una variable global puede hacerse accesible antes de su definición o en otro fichero fuente, utilizando el calificador **extern**. Esto quiere decir que el calificador **extern** se utiliza para hacer visible una variable global allí donde no lo era.

## Calificación de variables locales

Una variable local declarada como **auto** (variable automática) ([3], pág. 85) solamente es visible dentro del bloque donde está definida. Este tipo de variables no son iniciadas automáticamente, por lo que hay que iniciarlas explícitamente. Es recomendable iniciarlas siempre.

Una variable local declarada **static** solamente es visible dentro del bloque donde está definida; pero a diferencia de las automáticas su existencia es permanente, en lugar de aparecer y desaparcer al iniciar y finalizar la ejecución del bloque que la contiene.

Una variable local declarada **static** es iniciada solamente una vez, cuando se ejecuta el código que la define, y no es reiniciada cada vez que se ejecuta el bloque que la contiene, sino que la siguiente ejecución del bloque comienza con el valor que tenía la variable cuando finalizó la ejecución anterior. Si la variable no es iniciada explícitamente, C++ la inicia automáticamente a 0.

Una variable local declarada **register** es una recomendación al compilador para que almacene dicha variable, si es posible, en un registro de la máquina, lo que producirá programas más cortos y más rápidos. El número de registros utilizables para este tipo de variables depende de la máquina. Si no es posible almacenar una variable **register** en un registro, se le da el tratamiento de automática. Este tipo de declaración es válido para variables de tipo **int** y de tipo apuntador, debido al tamaño del registro.

Una variable declarada **extern** a nivel local hace referencia a una variable definida con el mismo nombre a nivel global en cualquier parte del programa. La finalidad de **extern** en este caso es hacer accesible una variable global en un lugar donde no lo es.

## Funciones recursivas

Como ya se mencionó, se dice que una función es recursiva si se llama a sí misma. El compilador de C++ permite cualquier número de llamadas recursivas a una función. Cada vez que la función es llamada, los parámetros formales y las variables **auto** y **register** son iniciadas. Notar que las variables **static** solamente son iniciadas una vez, en la primera llamada.

¿Cuándo es eficaz escribir una función recursiva? La respuesta es sencilla: cuando el proceso a programar sea por definición recursivo. Por ejemplo, el cálculo del factorial de un número, %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
n!=n((n-1)!)
\]
\end{document} es por definición un proceso recursivo que se enuncia así: %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
factorial(n)=n*factorial(n-1)
\]
\end{document}.

Por lo tanto, la forma idónea de programar este problema es implementando una función recursiva, Como ejemplo, a continuación, se muestra un programa que visualiza el factorial de un número. Para ello, se ha escrito una función factorial que recibe como parámetro un número entero positivo y devuelve como resultado el factorial de dicho número ([3], pág. 306).

%FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\begin{verbatim}
//Calculo del factorial de un n\'umero
#include <stdio.h>

unsigned long factorial(int n);

int main()
{
  int numero;
  unsigned long fac;
  do{
    printf("N\\'umero? ");
    scanf("%d",&numero);
  }while(numero < 0 || numero > 25);
  fac=factorial(numero);
  printf("El factorial de %d es %d\n",numero,fac);
  return 0;
}

unsigned long factorial(int n){
  if(n==0)
    return 1;
  else
    return n*factorial(n-1);
}
\end{verbatim}
\end{document}

REFERENCIAS

[1] Ceballos, Francisco Javier, “Enciclopedia del lenguaje C”, Octava reimpresión, Alfaomega Grupo Editor, México, mayo 2006, ISBN: 970-15-0333-3.

[2] Schildt, Herbert, “C Manual de referencia”, Cuarta edición, McGraw Hill/Interamericana de España, S.A.U., 2001. ISBN: 0-07-212124-6.

[3] Ceballos, Sierra Francisco Javier, “Enciclopedia del Lenguaje C++”, Segunda edición, Alfaomega Grupo Editor S.A. de C.V., 2009, ISBN: 978-607-7686-43-9.