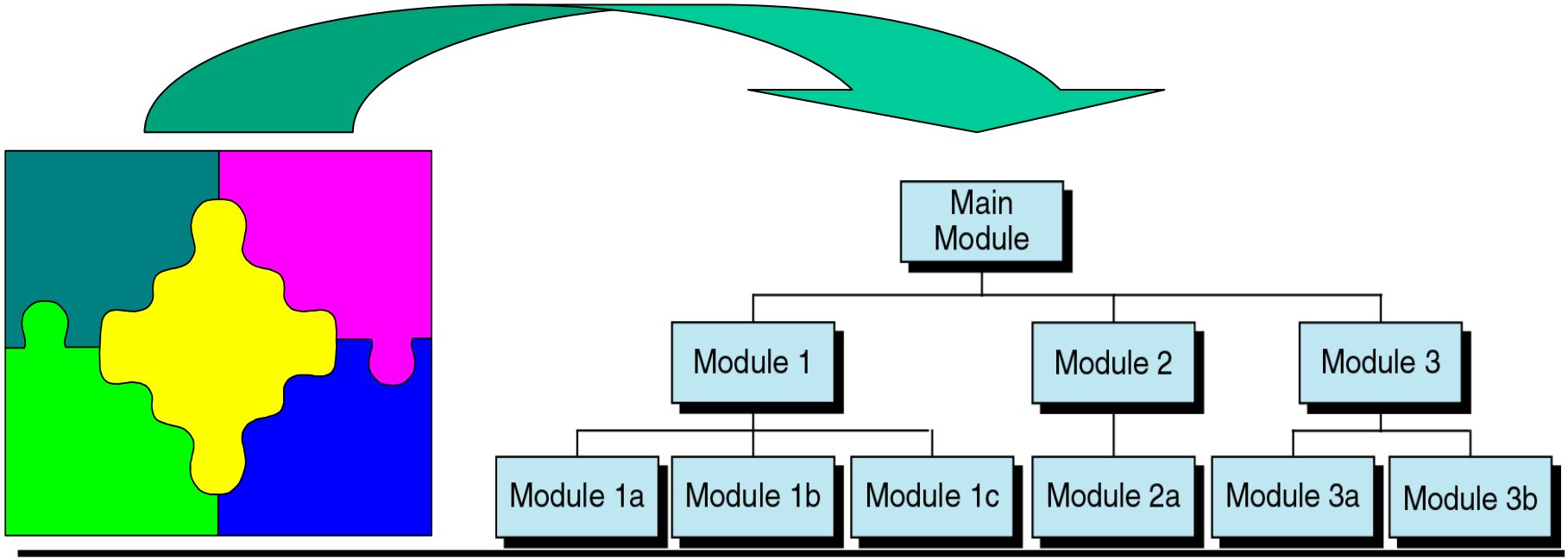
Funciones

Índice

- Estrategia de programación
- Uso y beneficios de las funciones.
- Sintaxis de la definición de una función y prototipado.
- Paso de argumentos a una función.
- Funciones y algoritmos recursivos.

Estrategia de programación: **dividir y vencer**

- Construir un programa a partir de pequeñas piezas o componentes.
- Cada pieza es más manejable que el programa original.



Estrategia de programación: **top - down**

Análisis y diseño en programación

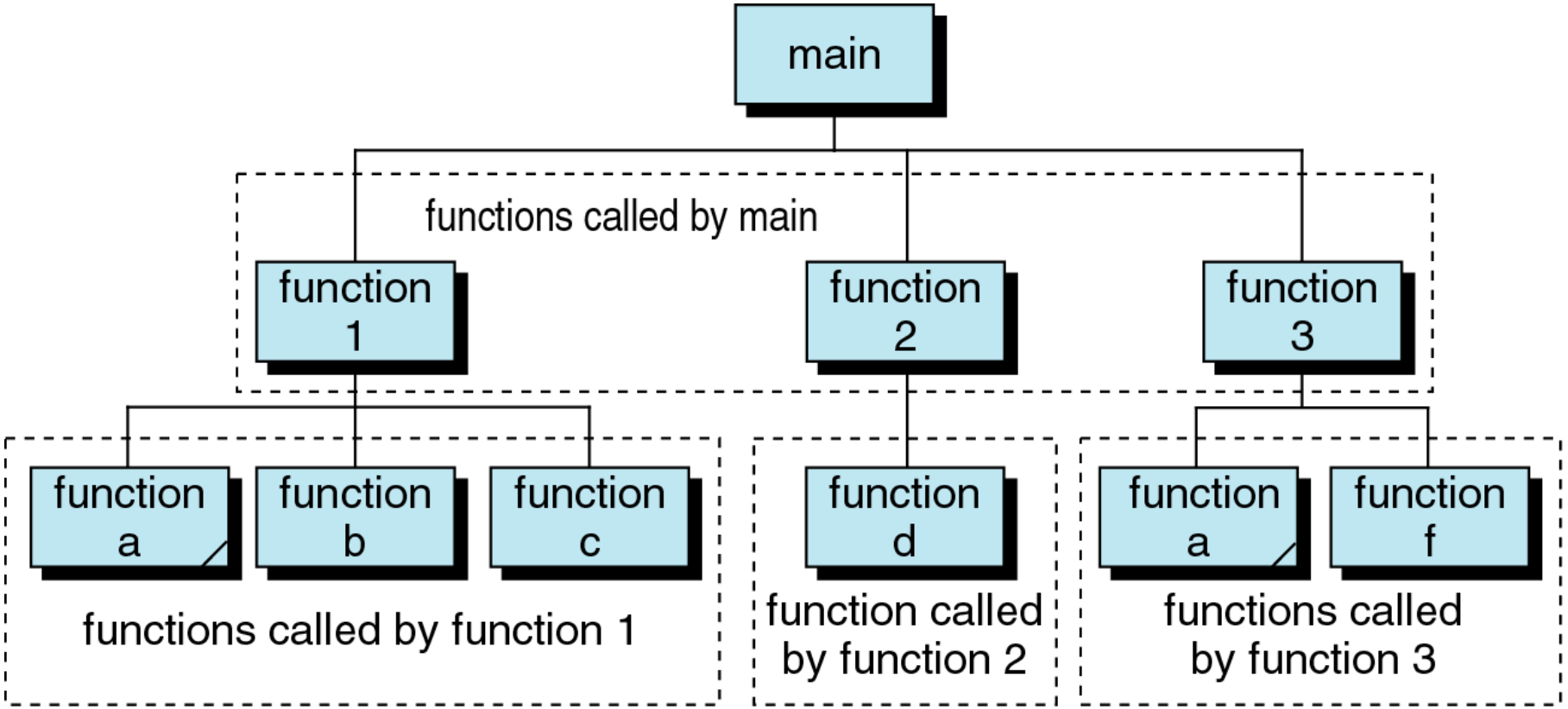
- Analizar el problema
- Diseñar una solución a grandes rasgos
- Una descomposición funcional muestra la forma de encajar las piezas
- Diseñar las funciones individuales

Descomposición funcional

- Buscar elementos comunes (similaridad)
- Parametrizar las características especiales (diferencias)
- Determinar qué funciones usarán otras
  - Usar un gráfico para mostrar las relaciones

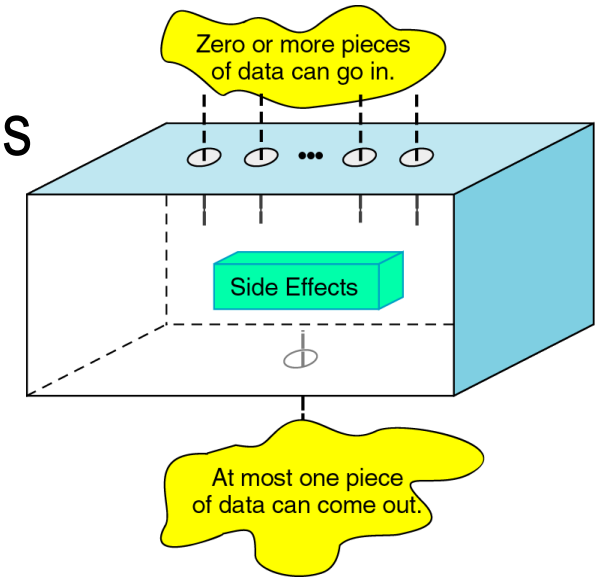
Estrategia de programación:
top - down

- Descomposición funcional de un programa



Características del uso de Funciones

- Modularizan un programa
- Todas las variables declaradas dentro de las funciones son variables locales
  - Se conocen sólo en la función definida
- Argumentos o Parámetros
  - Comunican información entre funciones
  - Variables locales



Beneficios del uso de Funciones

- La complejidad de cada función es mucho menor que la de todo el programa.
- Evitan repetición de código.
- Si se diseñan lo suficientemente generales, se pueden **reutilizar** (Reusabilidad del Software).
- Se puede repartir el trabajo entre varios programadores, reduciendo el tiempo de desarrollo.
- Se puede probar las funciones según se van terminando, sin necesidad de esperar a terminar todo el programa. Facilita la corrección de errores cometidos en las funciones.
- Facilita la depuración del programa, identificando y aislando la función incorrecta.
- Permite usar funciones creadas por otros programadores.

Beneficios del uso de Funciones

- Permiten reusar el código
  - Codificar una vez y usar varias veces
- Centralizar cambios
  - Cambios o detección de errores en un lugar
- Mejor organización de los programas
  - Fácil de probar, comprender y depurar
- Modularización para proyectos en equipo
  - Cada persona puede trabajar independientemente

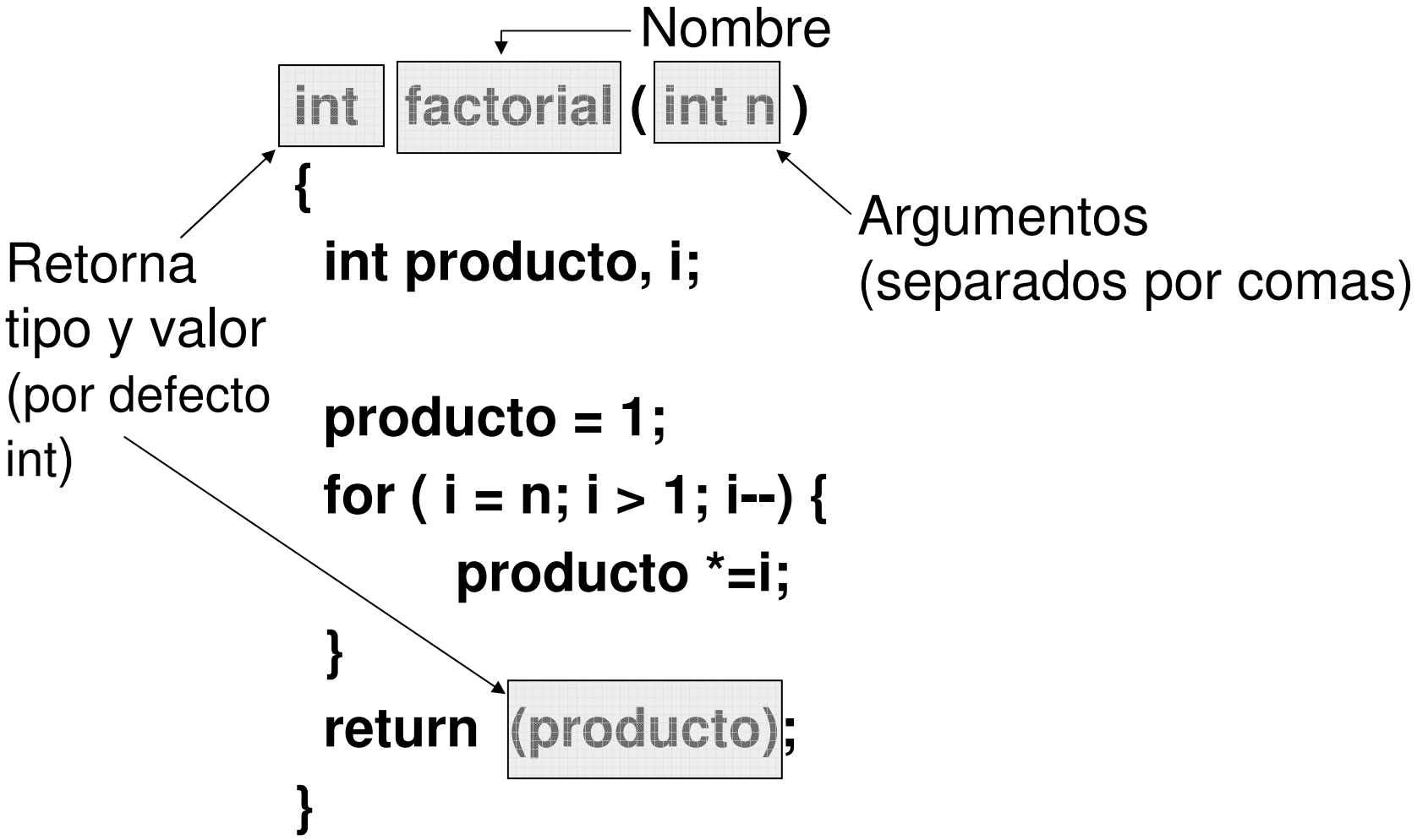
Sintaxis de la definición de una función

```
tipo_devuelto   nombreFuncion(tipo1 arg1,…, tipon argn)
{
        /* declaración de variables */;

          instrucción_1;
          instrucción_2;
                  …
          instrucción_n;

          return  expresión_devuelta;
}
```

Nota: void nombre_funcion() – no devuelve nada

nombre_funcion(void) – no recibe argumentos

Sintaxis de la definición de una función



Prototipo de una Función

- Para que una función pueda usarse en otras partes del programa es necesario colocar al principio de éste el **prototipo** de la función.
- La misión del prototipo es la de **declarar** la función al resto del programa, lo que permite al compilador:
  - Comprobar que los argumentos son correctos, tanto en número como en tipo.
  - Comprobar que el uso del valor devuelto por la función sea acorde con su tipo.
- Sintaxis del prototipo:

tipo_devuelto nombreFuncion(tipo1 arg1,…, tipo*n* arg*n*);

Ejemplo de uso de funciones

```
/*****************************************************************\
* Programa: factorial.c                                           *
* Descripción: Calcula el factorial de un numero usando una funcion *
*              con tipo de dato long y double                     *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>

long int factoriali (int n);
double factorialf (int n);

int main (void)
{
   int n;
   long int fact;

   printf ("Ingresar numero: ");
   scanf("%d", &n);
   fact = factoriali(n);
   printf("El factorial de %d es: %d\n",n, fact);
   printf("El factorial de %d es: %.0f\n",n, factorialf(n));
}
```

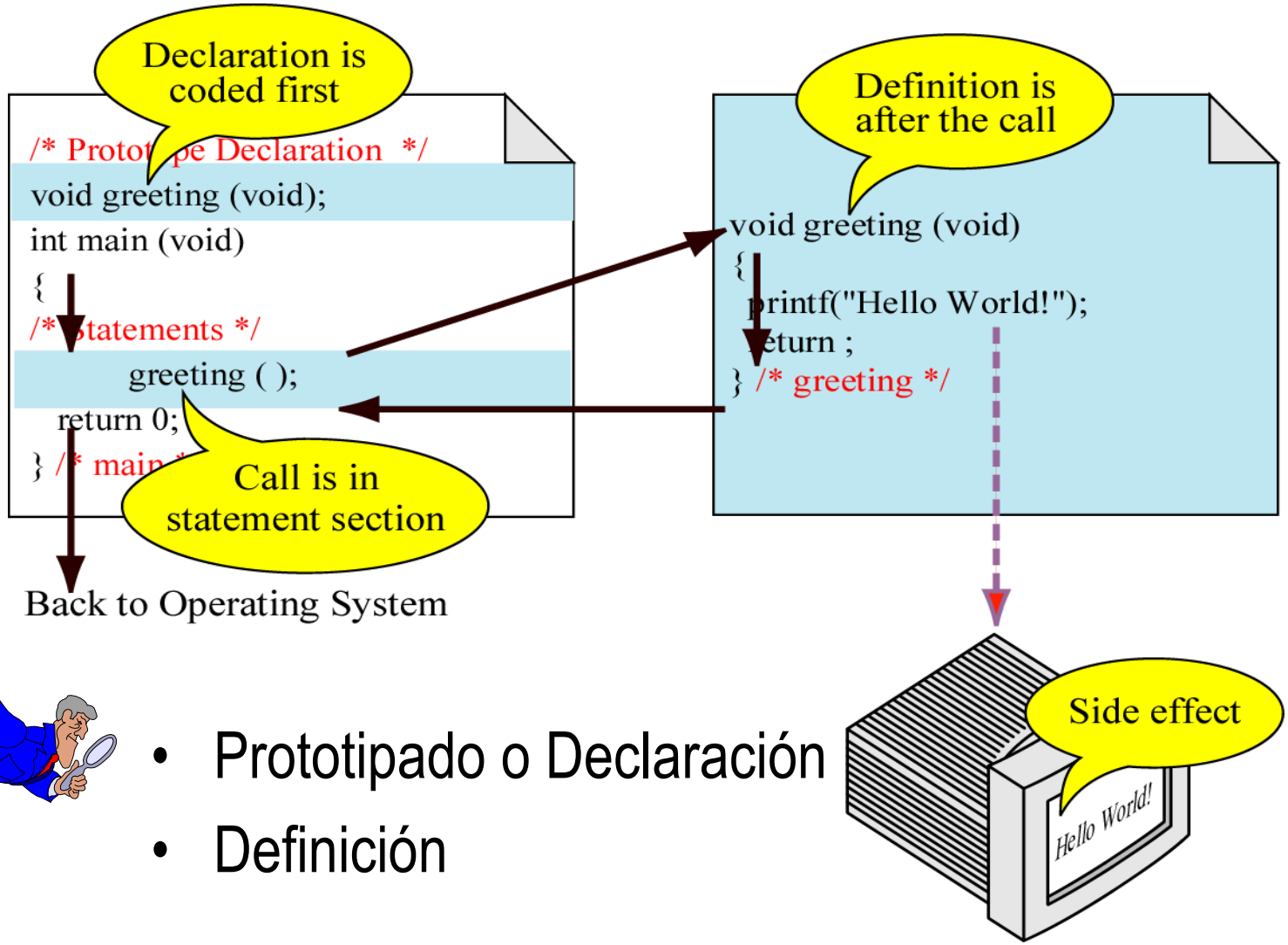
declaración de prototipos

Ejemplo de uso de funciones

```
long int factoriali ( int n )
{
   int i;
   long int producto;

   producto = 1;
   for ( i = n; i > 1; i--) {
        producto *=i;
   }
   return (producto);
}

double  factorialf ( int n )
{
   int i;
   double producto;

   producto = 1;
   for ( i = n; i > 1; i--) {
        producto *=i;
   }
   return (producto);
}
```

variables locales

Programas con funciones



- Prototipado o Declaración
- Definición

Paso de argumentos a una función

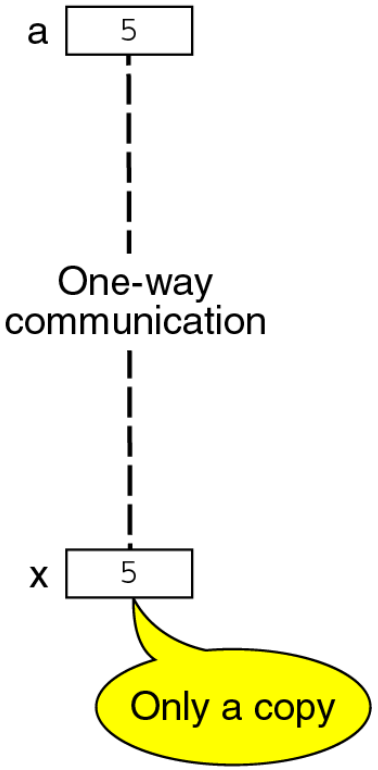
- Hay dos formas de pasar argumentos a una función:
  - Por valor.
  - Por referencia.
- **Paso de argumentos por valor**:
  - Cuando se pasa un valor a una función mediante un argumento, *se copia el valor* a la función. Por ello se puede modificar el valor del argumento dentro de la función, sin alterar el valor del argumento donde se realiza la llamada.
- En C el paso de argumentos es, por defecto, por valor.
- El paso por valor implica que la transferencia de información sólo se realiza en un sentido.

Paso de argumentos por valor

```c
/* Prototype Declarations */
void fun (int num1);

int main (void)
{
/* Local Definitions */
   int  a = 5;

/* Statements */
   fun (a)
   printf("%d\n", a);

   return 0;
}  /* main */
```

prints 5

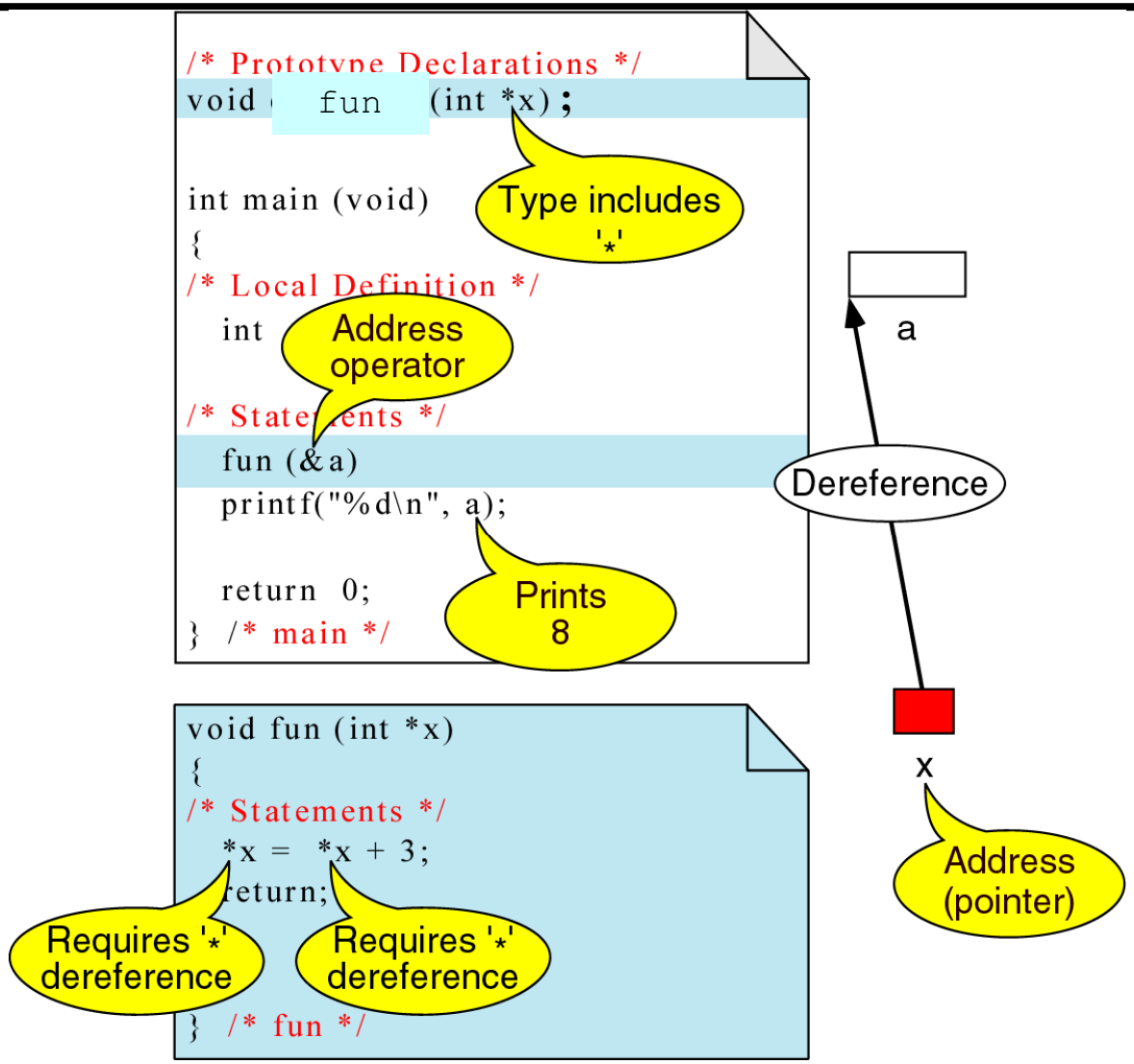
```c
void fun (int x)
{
/* Statements */
   x = x + 3;
   return;
}  /* fun*/
```



Paso de argumentos a una función

- **Paso de argumentos por referencia**:
  - Cuando se requiere que la función modifique el valor del argumento *real* que se pasa a una función se utiliza el paso por referencia. En este caso la modificación que se realice dentro de la función del argumento *formal* altera el valor del argumento real donde se realiza la llamada.
- El paso por referencia implica que la transferencia de información se realiza en ambos sentidos (entrada/salida).
- En este caso es necesario pasar la dirección de las variables y manejar su contenido con punteros.

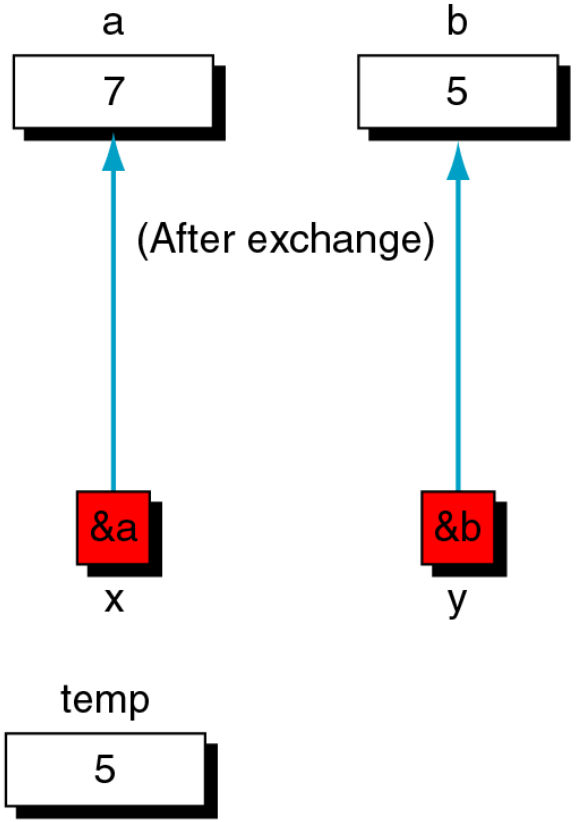
Paso de argumentos por referencia



Paso de argumentos por referencia

```c
/* Prototype Declarations */
void exchange (int *, int *);

int main (void)
{
  int  a = 5;
  int  b = 7;
  exchange (&a, &b);
  printf("%d %d\n", a, b);
  return 0;
} /* main */
```

```c
void exchange (int *x,
               int *y)
{
  int temp;

  temp = *x;
  *x   = *y;
  *y   = temp;
  return;
} /* exchange */
```



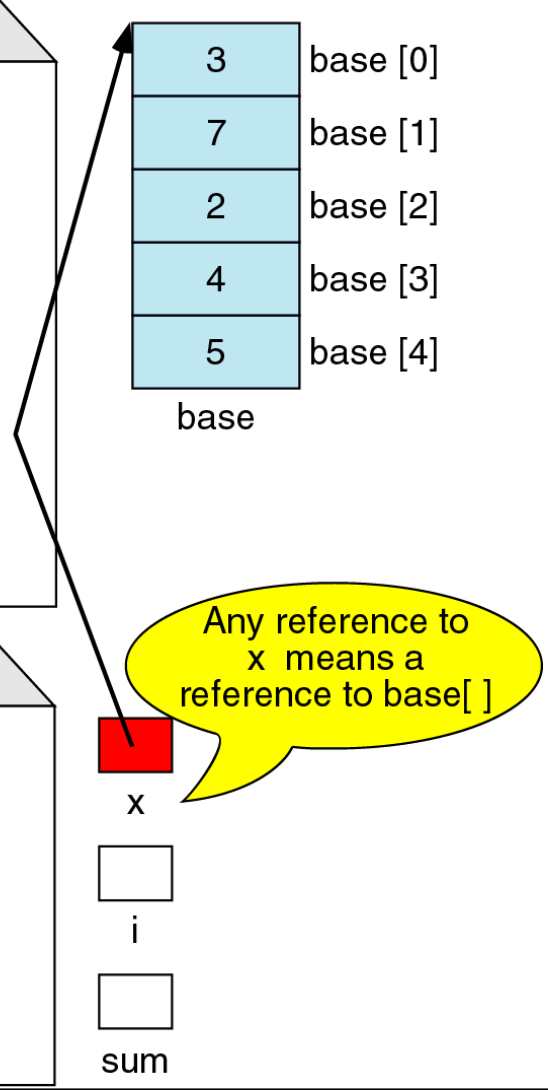
Paso de arrays a funciones

- Un array (vector) se pasa a una función poniendo el **nombre del array** sin corchetes ni índices.
- En el argumento formal se declara un array con un par de corchetes vacíos. Es decir, el tamaño del array no se especifica en la declaración de argumentos formales.
- En el caso de *arrays multidimensionales* las declaraciones de argumentos formales dentro de la definición de la función *deben* incluir especificaciones explícitas de tamaño en todos los índices *excepto el primero*.

Paso de arrays a funciones

```
#include <stdio.h>
/* Prototype Declarations */
double average (int x[]);

int main (void)
{
  double ave;
  int base[5] = {3, 7, 2, 4, 5};
  ...
  ave = average (base);
  ...
  return 0;
} /* main */
```

```
double average (int x [])
{
  int i;
  int sum = 0 ;
  for (i = 0; i < 5; i++)
      sum += x [i];
  return (sum / 5.0);
} /* main */
```



Paso de arrays a funciones

```c
#define MAX_ROWS  5
#define MAX_COLS  4
/* Prototype Declarations */
double average (int [][MAX_COLS]);
int main (void)
{
  double ave;
  int    table[MAX_ROWS][MAX_COLS] =
         {
          { 0, 1, 2, 3 },
          { 10, 11, 12, 13 },
          { 20, 21, 22, 23 },
          { 30, 31, 32, 33 },
          { 40, 41, 42, 43 }
          }; /* table */
  ...
  ave  =  average (table);
  ...
  return 0;
} /* main */
```

```c
double average (int x[][MAX_COLS])
{
  int    i;
  int    j;
  double sum = 0;
  for (i = 0; i < MAX_ROWS; i++)
       for (j = 0; j < MAX_COLS; j++)
            sum += x [i] [j];
   return(sum / (MAX_ROWS *  MAX_COLS);
} /* average */
```

Paso de estructuras a funciones

- Hay varias maneras de pasar información de una estructura a o desde una función.
- Se pueden transferir los miembros individuales o las estructuras completas.
- Una estructura completa se transfiere pasando la estructura como argumento (paso por *valor*) o un puntero a la estructura como argumento (paso por *referencia*).
- Una función puede devolver una estructura o un puntero a estructura.

Paso de punteros a funciones

- Se utilizan cuando se requiere pasar argumentos por *referencia*.
- Cuando se pasan variables simples normalmente se pasa como argumentos en la llamada de la función las direcciones de las variables (operador & precede al nombre de la variable).
- En la interfaz de las funciones se usan declaraciones de punteros y operaciones de indirección.
- Las modificaciones realizadas en las direcciones pasadas como argumentos se reconocen en la función y la rutina de llamada.

Paso de funciones a otras funciones

- Es posible pasar funciones como argumentos de otras funciones para aumentar la flexibilidad de un programa.
- En ese caso es necesario pasar un puntero a una función como argumento de otra función.
- La declaración del argumento formal es:

tipo_dato (*nombreFuncion) (tipo1 arg1,…, tipo*n* arg*n*)

- En la invocación se utiliza como valor cualquier identificador de función que tenga los mismos argumentos y resultado.
- El nombre de una función, de forma similar al nombre de un array, representa la dirección de inicio del código que define la función ().

Funciones recursivas

- *Recursividad* es una cualidad consistente en que una función se llama a sí misma de forma repetida, hasta que se satisface alguna condición determinada.
- Esta cualidad es necesaria para programar *algoritmos recursivos*.
- El ejemplo típico (más sencillo) de algoritmo recursivo es el cálculo del factorial de un número:

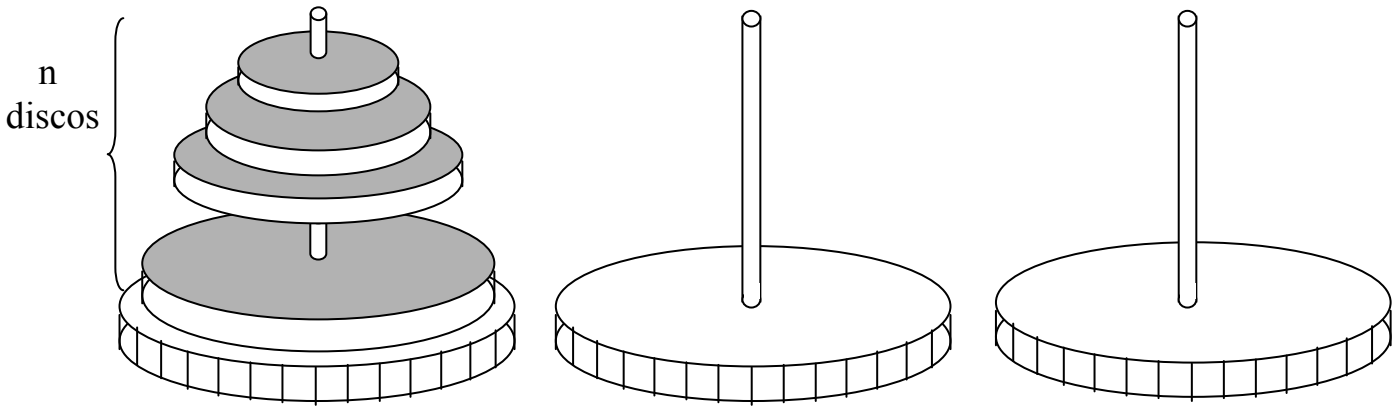
$$Fact(n) = \begin{cases} 1 & si\ n = 0 \\ n \cdot Fact(n-1) & si\ n > 0 \end{cases}$$

Algoritmos recursivos

- Para diseñar correctamente un algoritmo recursivo, es necesario:
  - Establecer correctamente la ley de recurrencia.
  - Definir el procedimiento de finalización del algoritmo recursivo(normalmente con el valor o valores iniciales).
- Para verificar funciones recursivas se aplica el método de las tres preguntas:
  - La pregunta Caso-Base: Hay una salida no recursiva de la función, y la rutina funciona correctamente para este caso “base”?
  - La pregunta Llamador-Más Pequeño: Cada llamada recursiva a la función se refiere a un caso más pequeño del problema original?
  - La pregunta Caso-General: Suponiendo que las llamadas recursivas funcionan correctamente, funciona correctamente toda la función?

Algoritmos recursivos: Torres de Hanoi

- Juego consistente en tres pivotes y un número de discos de diferentes tamaños apilados. El juego consiste en mover los discos desde un pivote, donde se encuentran inicialmente, a otro pivote, según las siguientes reglas:
  - Sólo se puede mover un disco cada vez
  - Un disco de mayor diámetro nunca puede estar encima de uno de menor diámetro



Torres de Hanoi: algoritmo

- La estrategia a seguir es considerar un pivote como origen y el otro como destino. El otro pivote se usa para almacenamiento auxiliar.
- Suponiendo n discos (>0), numerados del más pequeño al más grande, y que los pivotes toman el nombre detorre, atorre y auxtorre, el algoritmo para resolver el juego es:
  - Mover los n-1 discos superiores del pivote detorre al pivote auxtorre usando el pivote atorre como temporal.
  - Mover el disco n al pivote atorre.
  - Mover los n-1 discos del pivote auxtorre al pivote atorre usando detorre como temporal.

Torres de hanoi: programa

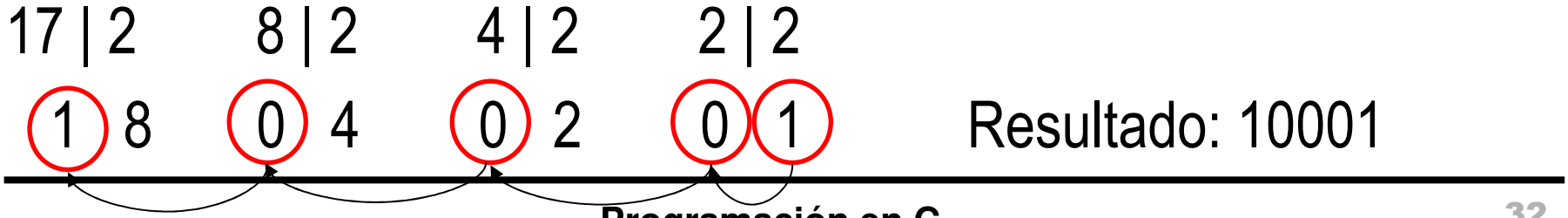
```
/*****************************************************************\
* Programa: thanoi.c                                              *
* Descripción: Programa que imprime los pasos para resolver el juego de  *
*              las torres de hanoi para n discos dados por el usuario    *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>
#define NUM_MAX_DISCOS 16

void torres_hanoi(int n, char detorre, char atorre, char auxtorre);

main(void)
{
 int n;

 printf("Torres de Hanoi: Cuantos discos? (max. %d) ",NUM_MAX_DISCOS);
 scanf("%d",&n);
 if (n > NUM_MAX_DISCOS)
 {
  printf("Numero de discos muy grande\n");
  return 1;
 }
 torres_hanoi(n,'A','C','B');
 return 0;
}
```

Torres de hanoi: función recursiva

```
/*****************************************************************\
* Funcion: torres_hanoi                                           *
* Descripción: imprime los pasos para resolver el juego de las torres de *
*               hanoi para n discos de manera recursiva           *
* Argumentos: int n: numero de discos                             *
*             char detorre: caracter del pivote donde estan los discos   *
*             char atorre: caracter del pivote destino            *
*             char auxtorre: caracter del pivote auxiliar         *
* Valor devuelto: ninguno (la funcion imprime los movimientos)    *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
void torres_hanoi(int n, char detorre, char atorre, char auxtorre)
{
 if (n > 0)
 {
  torres_hanoi(n-1,detorre,auxtorre,atorre);
  printf("mover disco %d de torre %c a torre %c\n",n,detorre,atorre);
  torres_hanoi(n-1,auxtorre,atorre,detorre);
 }
 return;
}
```

Recursividad: Cambio de base

- Se requiere imprimir un número entero en base decimal en otra base entre 2 a 16.
- El algoritmo para cambiar de base utiliza divisiones sucesivas hasta que el cociente sea menor que la base. A continuación se genera los dígitos del número resultante agrupando, de derecha a izquierda, el último cociente y los restos obtenidos durante la división desde el último al primero. Si los dígitos superan la base 10 se utilizan letras.
- Ejemplo: 17 en base 2.



Cambio de base: programa

```
/*****************************************************************\
* Programa: cambase.c                                             *
* Descripción: Programa que imprime el numero N en Base 2 <= Base <= 16  *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>

void  cambia_base( unsigned int N, unsigned int Base);

main()
{
  int i;

  i = 51;
  printf("%d  Base 10, ",i);
  printf("\t Base 2: ");
  cambia_base(i,2);
  printf("\t Base 8: ");
  cambia_base(i,8);
  printf("\t Base 16: ");
  cambia_base(i,16);
  printf("\n");
}
```

Cambio de base: función

```
/*****************************************************************\
* Funcion: cambia_base                                            *
* Descripción: imprime un numero entero N en la base Base 2 <= Base <= 16*
*              de manera recursiva                                *
* Argumentos: int N: numero decimal                               *
*             int Base: base a la que se desea convertir 2 <= Base <= 16 *
* Valor devuelto: ninguno (la funcion imprime el numero)          *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
void  cambia_base( unsigned int N, unsigned int Base )
{
 static char Tabla_Digitos[ ] = "0123456789abcdef";

 if( N >= Base )
    cambia_base( N / Base, Base  );
 putchar( Tabla_Digitos[ N % Base ] );
}
```

Características de las funciones recursivas

- La recursividad se realiza apilando los argumentos y variables locales, por lo que consume gran cantidad de memoria.
- Las sucesivas llamadas recursivas a la función consumen tiempo de CPU, por lo que el proceso es más **ineficiente** o más lento que uno iterativo.
- Todo algoritmo recursivo tiene su equivalente iterativo siendo éste más largo y costoso de desarrollar.
- Los algoritmos que tienen una base recursiva son más simples y elegantes de programar usando recursividad .

Estructura de un programa

Índice

- Tipos de almacenamiento.
- Variables automáticas.
- Variables estáticas.
- Variables externas o globales.
- Variables registro.
- Modificadores.

Tipos de almacenamiento

- Las variables se caracterizan por su tipo de datos y por su tipo de almacenamiento.
- El tipo de almacenamiento se refiere a la *permanencia* de la variable y a su *ámbito* dentro del programa (parte).
- La permanencia o duración de una variable se refiere a la gestión temporal del almacenamiento en memoria de una variable.
- El ámbito o alcance de una variable se refiere a la región del programa donde se reconoce la variable.
- C dispone de cuatro especificadores de tipos de almacenamiento: auto, static, extern y register y dos modificadores de tipo de almacenamiento (const y volatile).

Variables automáticas

- Se declaran siempre dentro de la función o bloque de instrucciones ({ }) anteponiendo a la declaración la palabra **auto**. Por defecto, cualquier variable local declarada dentro de una función se considera como automática.
- La duración de una variable automática se conserva mientras se ejecuta la función o bloque donde se ha declarado. Cuando termina su alcance, el compilador libera esa posición de memoria perdiéndose su valor. Si se reingresa al alcance de la variable se asigna una nueva dirección.
- El ámbito de una variable automática es local a la función o bloque donde se ha declarado.

Variables automáticas

- Las variables automáticas cuando se declaran e inicializan son reinicializadas cada vez que se ingresa a su bloque de alcance.

```
#include <stdio.h>
void incremento(void)
{ auto int j=1;
  j++; printf( "j: %d\n", j );
}
int main( void )
{ incremento();
  incremento();
  incremento();
}
```

2
2
2

Variables estáticas

- Se declaran siempre dentro de la función o bloque de instrucciones ({ }) anteponiendo a la declaración la palabra **static**.
- El ámbito de una variable estática es local a la función o bloque donde se ha declarado.
- La duración de una variable estática se extiende mientras se ejecuta el programa. Es decir *retienen* sus valores durante toda la vida del programa. Esta característica permite a las funciones mantener información a lo largo de la ejecución del programa.

Variables estáticas

- Las variables estáticas se pueden inicializar mediante una constante al momento de su declaración. Si no se inicializan se asignan el valor 0.

```
#include <stdio.h>
void incremento(void)
{ static int k=1;
  k++; printf( “k: %d\n", k );
}
int main( void )
{ incremento();
  incremento();
  incremento();
}
```

2
3
4

Variables externas o globales

- Las variables externas (ve) se *definen* y se *declaran*. La definición de una ve se escribe de la misma manera que una declaración de una variable ordinaria, fuera y normalmente antes de las funciones que acceden a ellas y son considearadas estáticas. La *declaración* de una ve empieza por el especificador **extern**.
- El ámbito de una variable externa se extiende desde el punto de su definición hasta el resto del programa, pudiendo abarcar dos o más funciones e incluso varios archivos.
- La duración de una variable externa es similar a una estática.

Variables externas

- Las ve proporcionan un mecanismo adicional de transferencia de información entre funciones sin usar argumentos.

Primer archivo:

```
extern int a, b, c; /* declaración de vars. externas */
void func1(void)
{    . . .
}
```

Segundo archivo:

```
int a=1, b=2, c=3; /* definición de vars. externas */
main( )
{    . . .
}
```

Variables registro

- Las variables registro son áreas especiales de almacenamiento dentro de la CPU cuyo acceso es muy rápido. Algunos programas pueden reducir su tiempo de ejecución, si algunos valores pueden almacenarse dentro de los registros en vez de la memoria del computador.
- La declaración de una variable registro empieza por el especificador **register**. Éste recomienda al compilador que la variable debe mantenerse en un registro para ganar velocidad de procesamiento.
- El ámbito y duración de una variable register es similar a una automática.

Variables registro

- Como los procesadores disponen de un número limitado de registros se recomienda un uso moderado del especificador *register*. El operador dirección (&) **no** se aplica a las variables registro. Normalmente sólo variables enteras se declaran *register*.

```
int strlen( register char *p)
{
  register int len = 0;
  while (*p++)
    len++;
  return len;
}
```

Modificadores const y volatile

- **const** informa al compilador que la variable no puede modificarse.
- **volatile** informa al compilador que la variable puede modificarse por medios ajenos al compilador C (drivers, controladores via hardware).