Programación avanzada

Pedro Corcuera

Dpto. Matemática Aplicada y
Ciencias de la Computación

Universidad de Cantabria

corcuerp@unican.es

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).

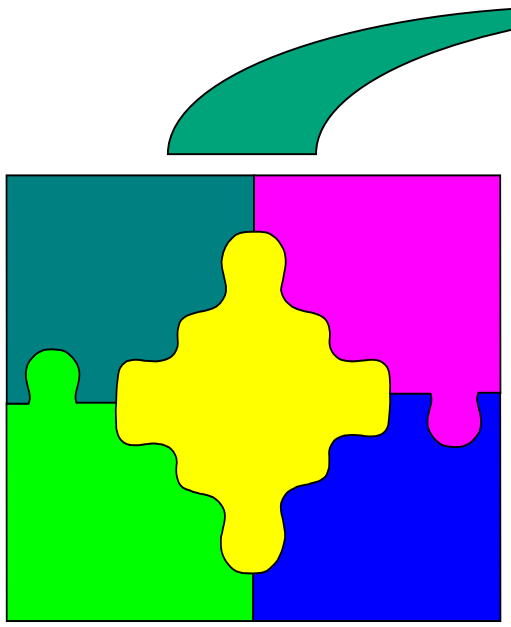
Funciones

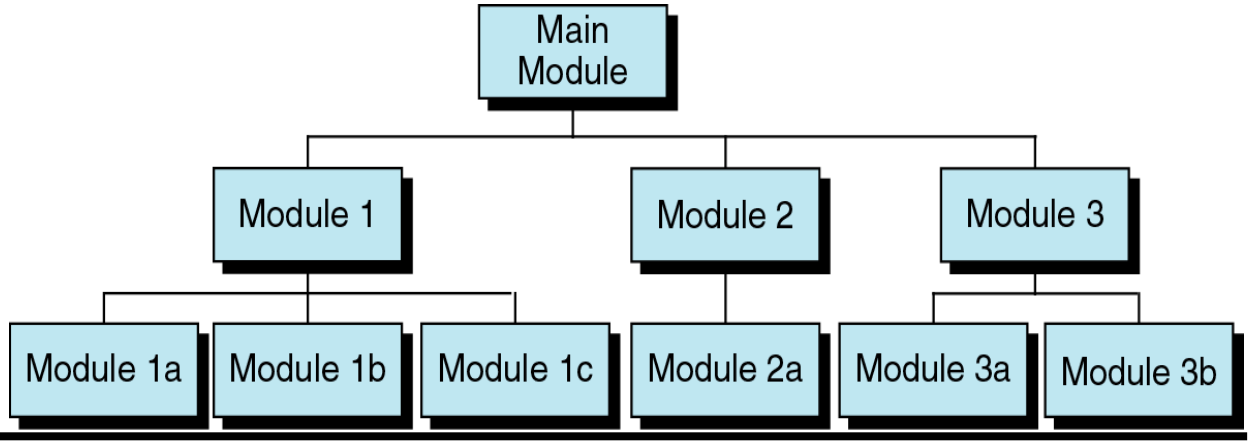
Índice

- 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

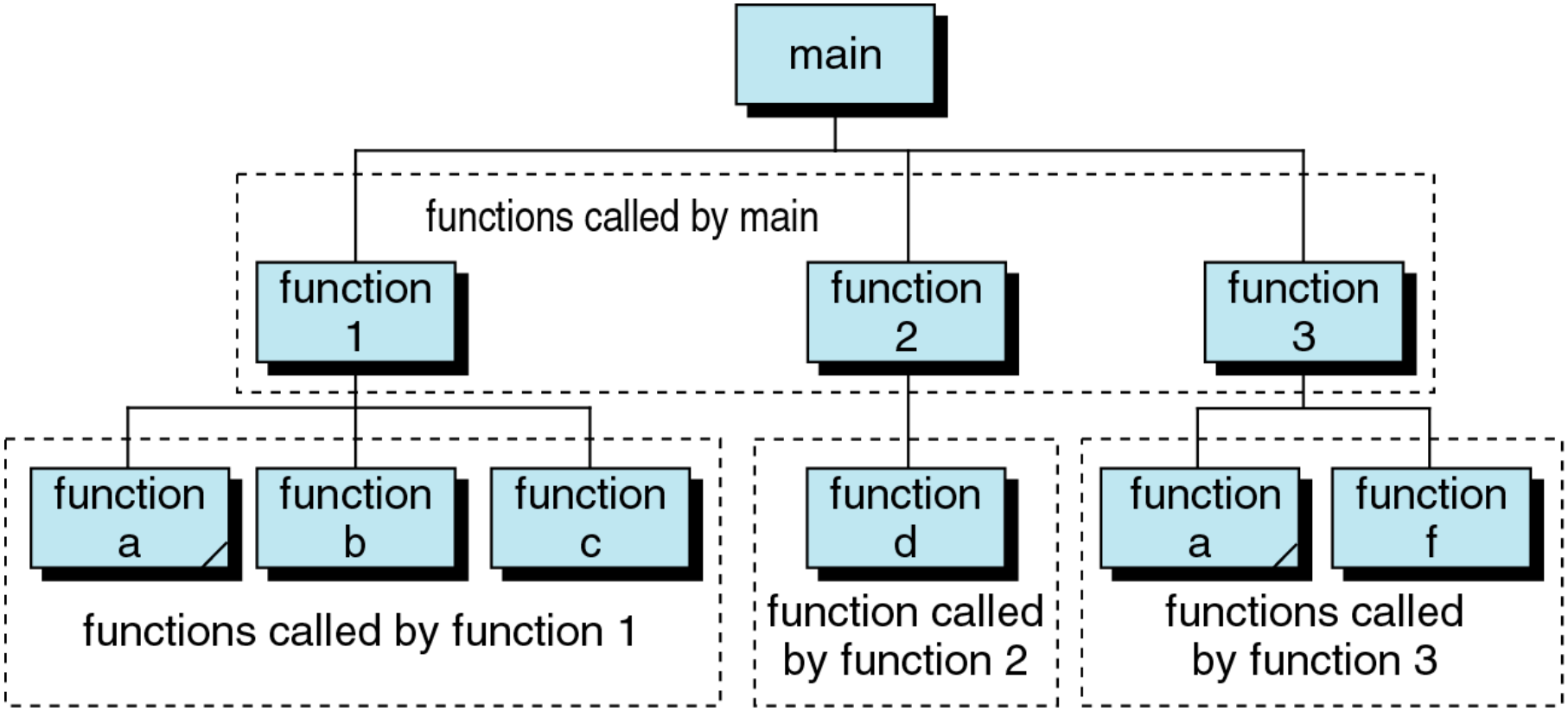
- Analizar el problema
- Diseñar una solución a grandes rasgos
- Una descomposición funcional muestra la forma de encajar las piezas
- Diseñar las funciones individuales

Descomposición funcional

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

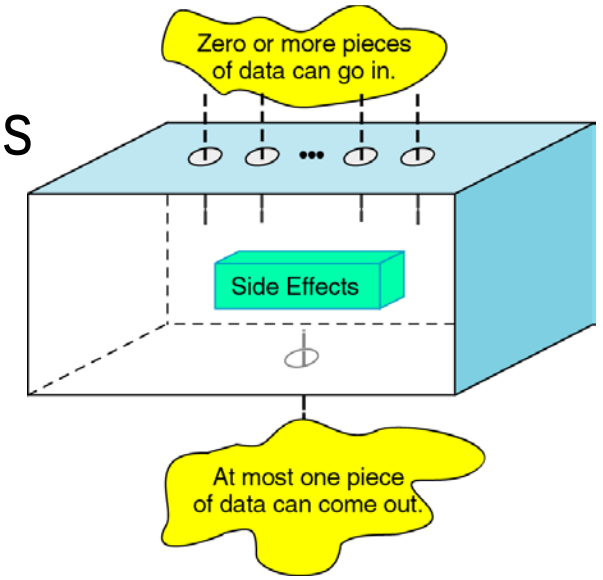
Estrategia de programación: **top - down**

- 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

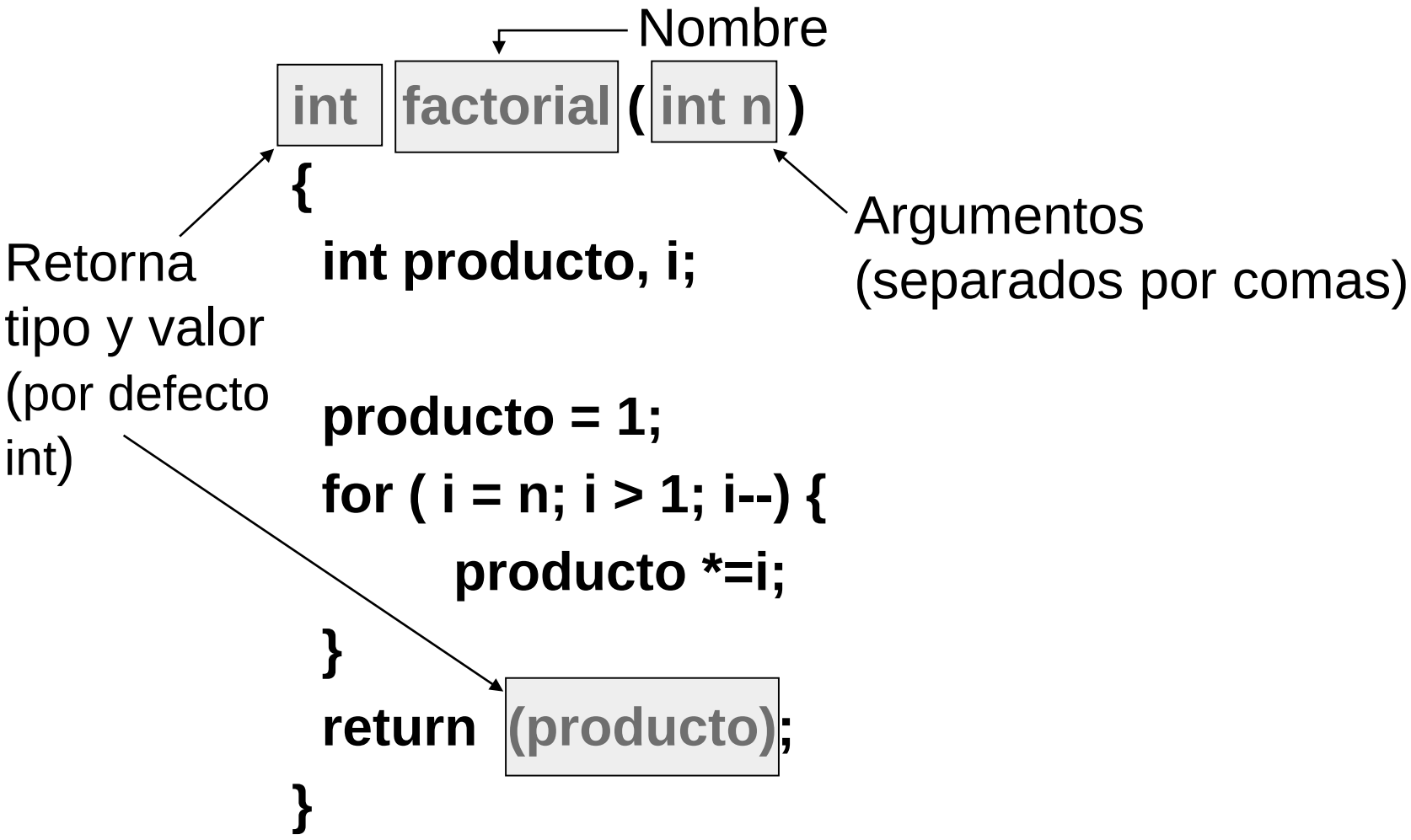
Sintaxis de la definición de una función

```
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));
}
```

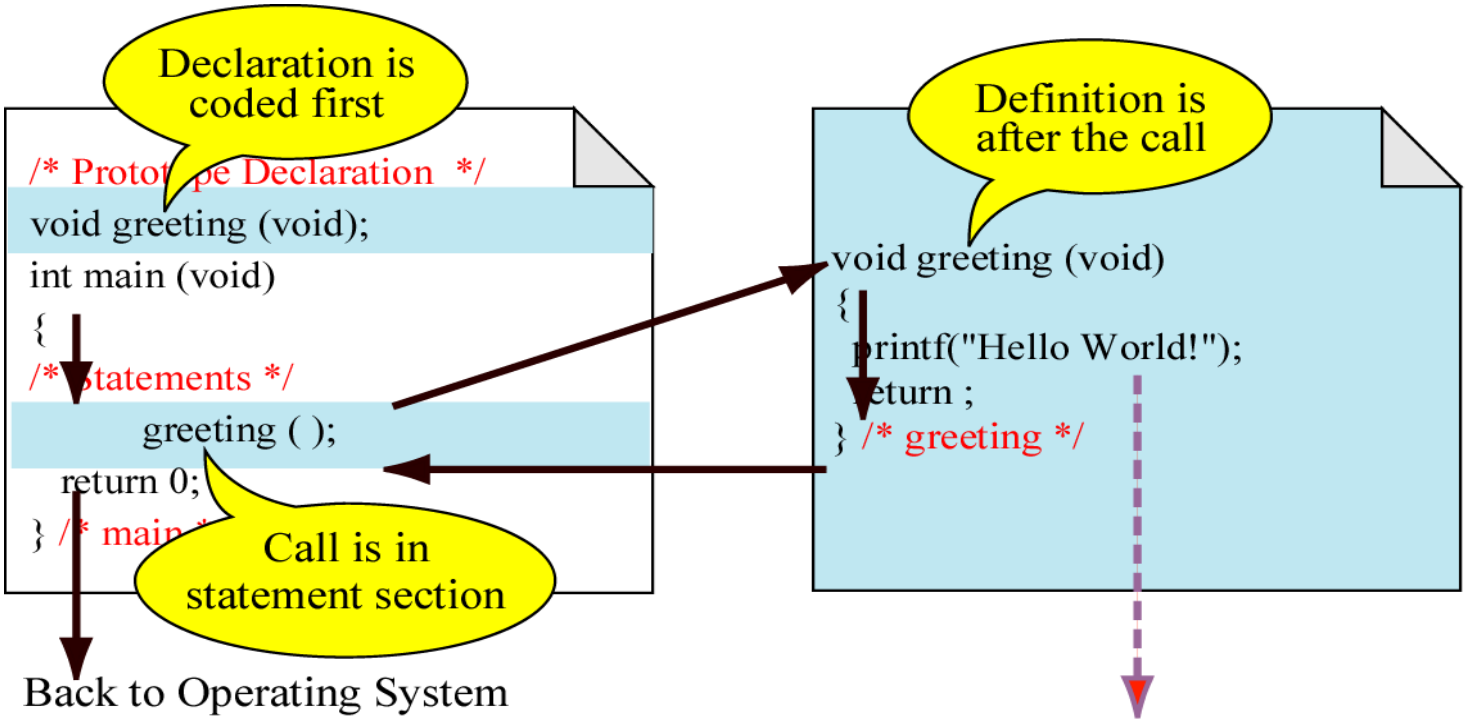
declaración de prototipos

Ejemplo de uso de funciones

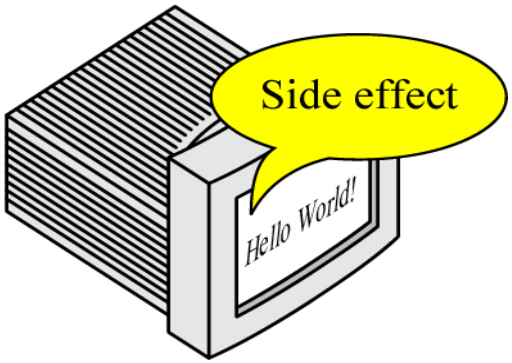
```
long int factoriali ( int n )
{
   int i;
   long int producto;

   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

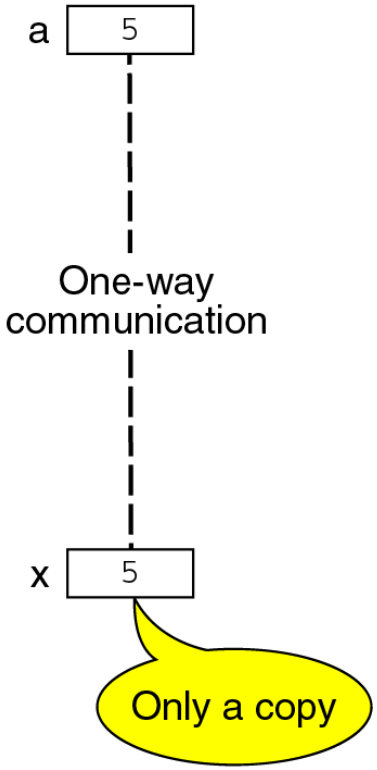
- Hay dos formas de pasar argumentos a una función:
  - Por valor.
  - Por referencia.
- Paso de argumentos por valor:
  - Cuando se pasa un valor a una función mediante un argumento, *se copia el valor* a la función. Por ello se puede modificar el valor del argumento dentro de la función, sin alterar 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

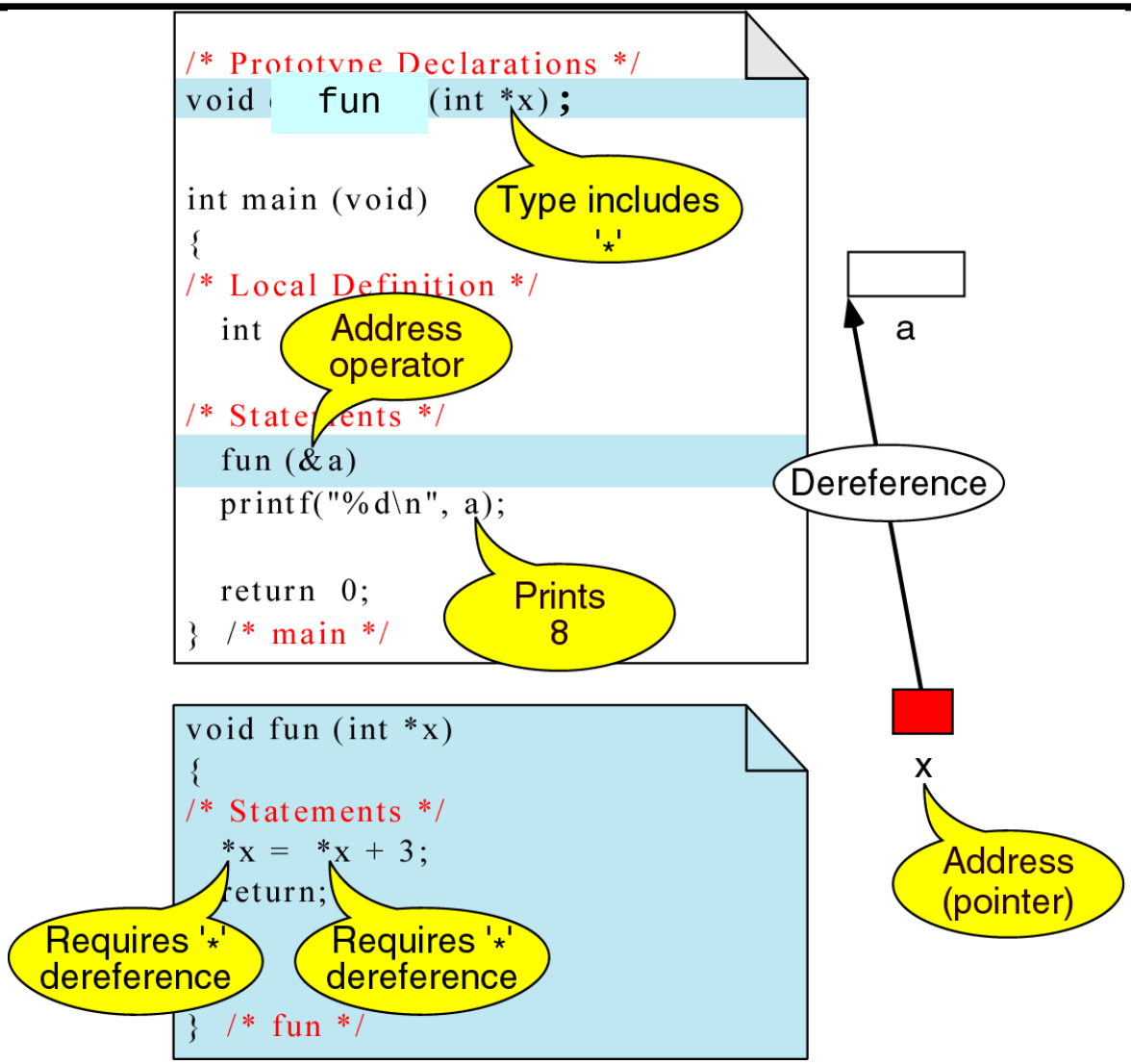
```c
void fun (int x)
{
/* Statements */
   x = x + 3;
   return;
}  /* fun*/
```



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.

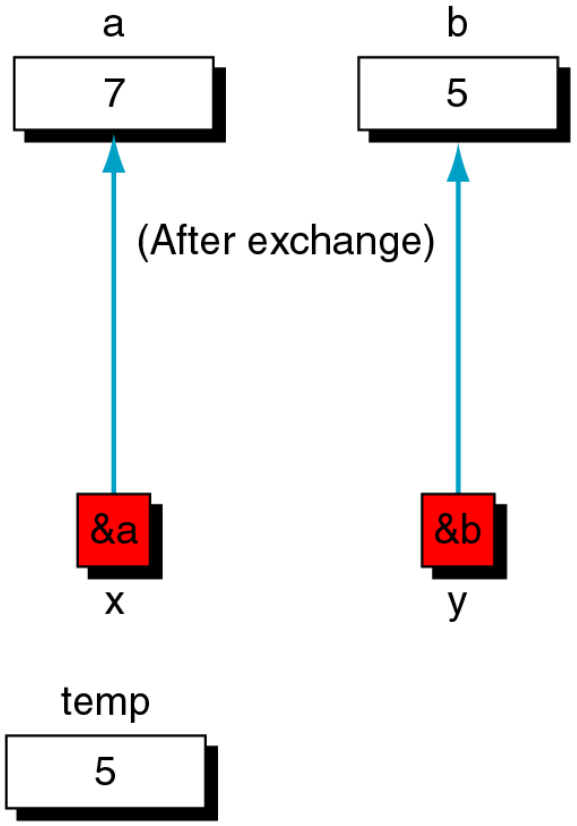
Paso de argumentos por referencia



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 */
```



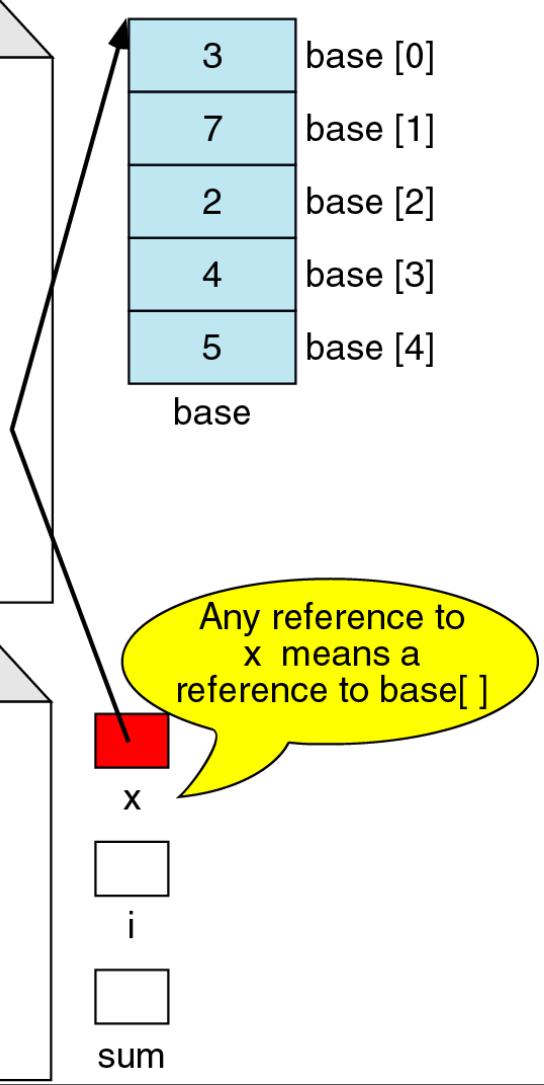
Paso de arrays a funciones

- 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

```c
#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 */
```

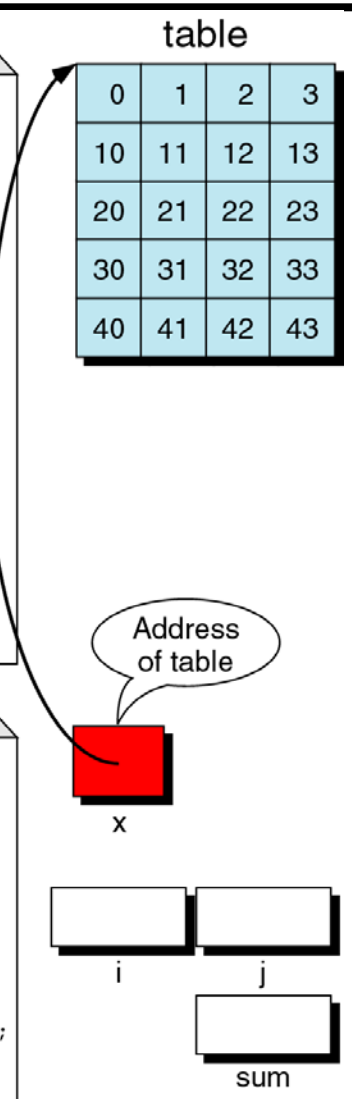
```c
double average (int x [])
{
  int i;
  int sum = 0 ;
  for (i = 0; i < 5; i++)
      sum += x [i];
  return (sum / 5.0);
} /* main */
```



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:

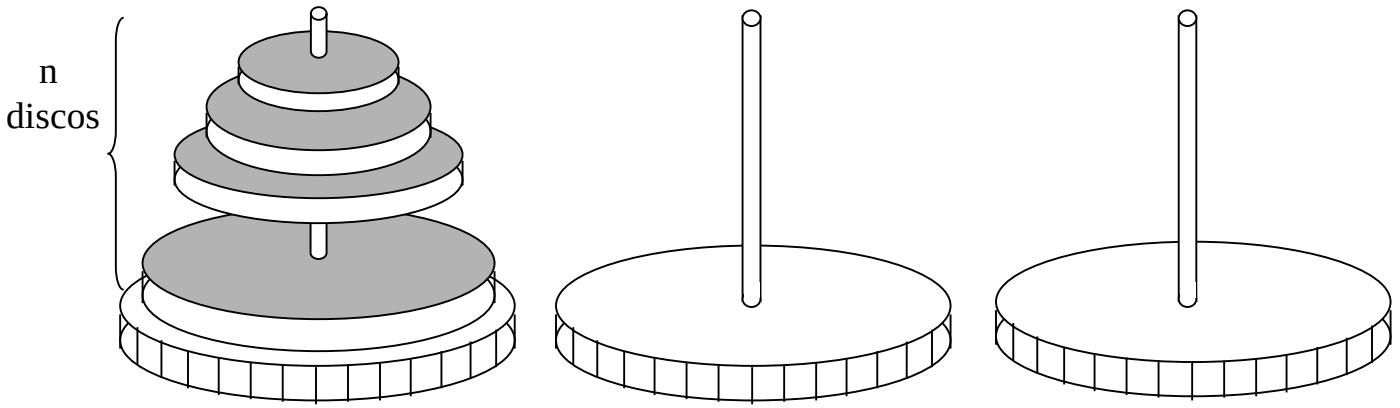
$$Fact(n) = \begin{cases} 1 & si\ n = 0 \\ n \cdot Fact(n-1) & si\ n > 0 \end{cases}$$

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

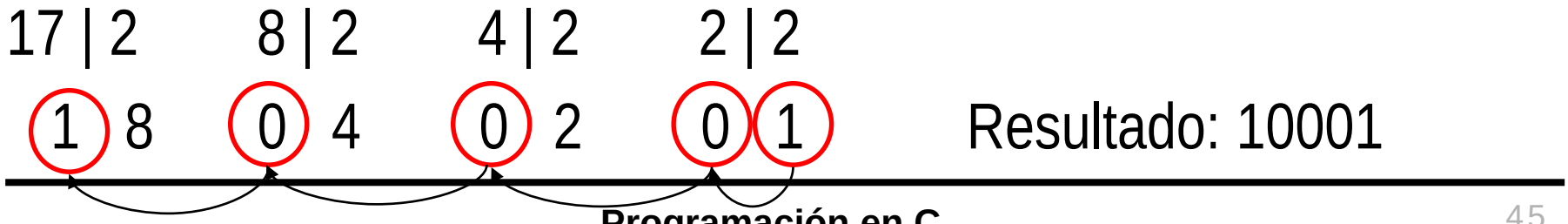
```
/*****************************************************************\
* Programa: thanoi.c                                              *
* Descripción: Programa que imprime los pasos para resolver el juego de  *
*              las torres de hanoi para n discos dados por el usuario    *
* Autor: Pedro 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 .

Ordenación rápida (Quicksort)

- La ordenación rápida se basa en el hecho que los intercambios deben ser realizados preferentemente sobre distancias grandes.
- El algoritmo a seguir es el mismo que se aplica cuando se quiere ordenar un gran montón de exámenes: Seleccionar un valor de división (L por ejemplo) y dividir el montón en dos pilas, A-L y M-Z. Después se toma la primera pila y se subdivide en dos, A-F y G-L por ejemplo. A su vez la pila A-F puede subdividirse en A-C y D-F. Este proceso continúa hasta que las pilas sean suficientemente pequeñas para ordenarlas fácilmente. El mismo proceso se aplica a la otra pila.

Ordenación rápida (Quicksort)

- En la ordenación rápida se toma un elemento x del array (el del medio por ejemplo),
- Se busca en el array desde la izquierda hasta que $a_1, a_2, \ldots, a_{i-1} > x$, lo mismo se hace desde la derecha hasta encontrar $a_1, a_2, \ldots, a_{i-1} < x$.
- Después se intercambia esos elementos y se continúa ese proceso hasta que los índices se cruzan.
- Se aplica el mismo proceso para la porción izquierda del array entre el extremo izquierdo y el índice derecho y para la porción derecha entre el extremo derecho y el último índice izquierdo (algoritmo recursivo).

Ordenación rápida (Quicksort)

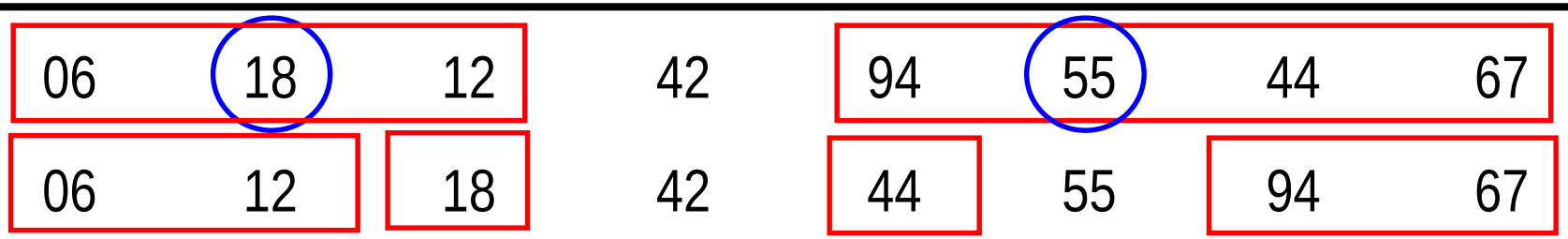
- La complejidad del algoritmo es O(n log n).
- Una comparación visual de los tiempos de ejecución de los algoritmos de ordenación se encuentra en:

http://personales.unican.es/corcuerp/ProgComp/Ordena/AlgoritmosOrdenamiento.html

Ordenación rápida

44	55	12	42	94	18	06	67
44	55	12	42	94	18	06	67
06	55	12	42	94	18	44	67
06	18	12	42	94	55	44	67
06	18	12	42	94	55	44	67

Ordenación rápida



Ordenación rápida

06	12	18	42	44	55	94	67
06	12	18	42	44	55	67	94

Ordenación rápida

```
/*************************************************************************\
* Programa: quicksort.c                                                  *
* Descripción: Programa que ordena un array mediante el metodo rapido    *
* Autor: Pedro Corcuera                                                  *
* Revisión: 1.0 2/02/2008                                                *
\*************************************************************************/
#include <stdio.h>

void qsort(int izq, int der, float a[]);

main()
{
 float notas[]={10.0,8.0,5.5,3.4,3.2,2.5,2.2,1.5};
 int i,n;


 n = sizeof(notas)/sizeof (float);

 qsort(0,n-1,notas);

 for(i = 0; i < n; i++)
  printf("%d  %10.2f \n",i,notas[i]);
 return 0;
}
```

Ordenación rápida

```
void qsort(int izq, int der, float a[])
{
 int i,j;
 float tmp,x;

 i = izq;
 j = der;
 x = a[(izq + der)/2];
 do
 {
  while (a[i] < x)  i++;
  while (x < a[j])  j--;
  if (i <= j)
  {
   tmp = a[i];
   a[i] = a[j];
   a[j] = tmp;
   i++;
   j--;
  }
 }
 while (i <= j);
 if (izq < j)
   qsort(izq, j, a);
 if (i < der)
   qsort(i, der, a);
}
```

Algoritmos de vuelta atrás (backtracking)

- Este tipo de algoritmos trata de encontrar soluciones de problemas específicos sin seguir una regla fija de computación, sino mediante *prueba y error*.
- El patrón común es descomponer el proceso de prueba y error en tareas parciales. Normalmente esas tareas se expresan en términos recursivos y consisten en la exploración de un número finito de sub-tareas.
- El proceso global como un proceso de búsqueda que construye progresivamente y poda un árbol de subtareas.
- Para reducir el tiempo de computación se usan heurísticas para la poda del árbol.

Algoritmos de vuelta atrás (backtracking)

- La técnica consiste en recorrer sistemáticamente todos los posibles caminos. Cuando un camino no conduce a la solución, se retrocede al paso anterior para buscar otro camino (backtracking).
- Si existe solución es seguro que se encuentra, el problema es el tiempo de proceso.
- Es deseable que sólo recorra los caminos que conduzcan a la solución y no todos. Es posible hacerlo cuando estando en un punto nos damos cuenta que si seguimos avanzando no encontraremos ninguna solución.
- Es un técnica netamente recursiva.

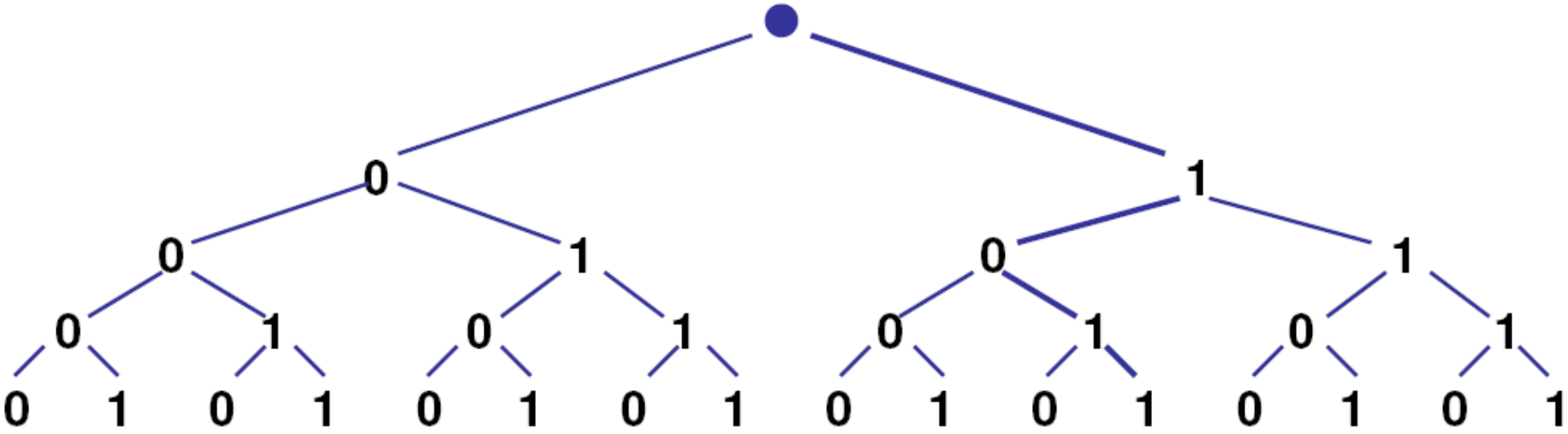
Backtracking – Selección de objetos

- Se tienen n objetos, cada uno puede o no ser seleccionado, ¿Cuáles son todas las formas posibles de tomarlos? ¿De cuántas maneras los podemos escoger?

Estrategia de solución

- Primero pensemos como será nuestro árbol:
  - En cada nivel podemos decidir si elegir o no el artículo.
  - La cantidad de decisiones que se deben tomar depende de la cantidad de artículos.
  - Dicha cantidad de decisiones es la profundidad del árbol.

Backtracking – Selección de objetos



- Arbol para cuatro objetos. Por ejemplo el camino 1011 significa que se toman el primer, tercer y cuarto artículos y no se toma el segundo.
- El número de posibles selecciones es 2^n

Selección de objetos

```
/*****************************************************************\
* Programa: selecc_obj.c                                          *
* Descripción: Programa para seleccionar un conjunto de n elementos      *
*              (0=no  1=si)                                       *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>
#define MAX 20
#define FALSE 0
#define TRUE  1

void escoger(int, int, int [ ]);
void imprimir(int, int[ ]);

void main(void)
{
 int i, n;
 int decision[MAX];
 printf("Nro de elementos: ");
 scanf("%d",&n);

// marcamos objetos como no escogidos
 for(i=0; i<n; i++)
   decision[i]=FALSE;
 escoger(0, n, decision);
}
```

Selección de objetos

```
void imprimir(int n, int decision[ ])
{
  int i;
  for(i=0; i<n; i++)
    printf("%d ", decision[i]);
  printf("\n");
}

void escoger (int k, int n, int decision[ ])
{
  if (k<n)
  {
   // no se toma el objeto
   decision[k]= FALSE;
   escoger(k+1, n, decision);
   // se toma el objeto
   decision[k]= TRUE;
   escoger(k+1, n, decision);
   decision[k]= FALSE;
  }
  else
    imprimir(n, decision);
}
```

Backtracking – Esquema general

- Como hay que recorrer un árbol y en cada paso se necesita encontrar todas las alternativas posibles y luego recorrerlas una a una, podemos pensar en el esquema siguiente:
  - Si se llega a la máxima profundidad del árbol, entonces mostrar solución (caso base).
  - Sino entonces:
    - Construir cada candidato
    - Para cada candidato añadirlo a la solución y llamar recursivamente a la función con el nivel siguiente de profundidad.

Backtracking – Esquema general

- Lo que cambia en cada problema es:
  - La forma de generar cada candidato.
  - Los tipos de datos.
  - El número de parámetros de la funciones.
  - Añadirle condiciones adicionales propias del problema.
  - Añadirle condiciones adicionales para reducir el árbol (podar).

- Reescribiendo la solución del problema de selección de objetos a este esquema general:

Selección de objetos – esquema general

```
/*************************************************************************\
* Programa: selecc_obj.c                                                 *
* Descripción: Programa para seleccionar un conjunto de n elementos      *
*              (0=no  1=si) segun esquema general                        *
* Autor: Pedro Corcuera                                                  *
* Revisión: 1.0 2/02/2008                                                *
\*************************************************************************/
# include <stdio.h>
# define MAX 20
#define FALSE 0
#define TRUE  1

void escoger(int, int, int [ ]);
void imprimir(int, int[ ]);
int construyeCandidatos(int [ ]);

void main(void)
{
 int i, n;
 int decision[MAX];
 printf("Nro de elementos: ");
 scanf("%d",&n);

// marcamos objetos como no escogidos
 for(i=0; i<n; i++)
   decision[i]=FALSE;
 escoger(0, n, decision);
}
```

Selección de objetos – esquema general

```
void imprimir(int n, int decision[ ])
{
  int i;
  for(i=0; i<n; i++)
    printf("%d ", decision[i]);
  printf("\n");
}
void escoger (int k, int n, int decision[ ])
{
  int cand[MAX];

  if (k == n)
     imprimir(n, decision);
  else
  {
    int i, ncand = construyeCandidatos(cand);
    for(i=0; i < ncand; i++)
    {
      decision[k] = cand[i];
      escoger(k+1, n, decision);
    }
  }
}
int construyeCandidatos(int decision[])
{
  decision[0]=TRUE; decision[1]=FALSE;
  return 2;
}
```

Variaciones

- Las variaciones o permutacione, son las distintas formas en que pueden agruparse r elementos de un conjunto n donde r < n. Deben diferir en algún elemento o en el orden.
- El número de variaciones en esas condiciones es: $V_r^n = \frac{n!}{(n-r)!}$
- Ejemplo: si n = 4 (abcd) y r = 3 entonces el número de variaciones es 24

abc	bac	cab	dab
abd	bad	cad	dac
acb	bca	cba	dba
acd	bcd	cbd	dbc
adb	bda	cda	dca
adc	bdc	cdb	dcb

Números que suman un valor objetivo

- Dado un conjunto de números enteros {2, 3, 6, 1, 5}, encontrar los subconjuntos cuya suma sea exactamente 9. Por ejemplo: {3, 1, 5} suman 9.

El Problema de las N Reinas

- En un tablero de ajedrez (NxN) colocar N reinas sin que se produzcan conflictos por el movimiento propio de las reinas.
- Ejm: N = 8

El Problema de las N Reinas

Problema de la mochila

- Supóngase que se tiene n objetos distintos y una mochila.
- Cada objeto tiene asociado un peso w_i y un valor v_i
- La mochila puede llevar un peso que no sobrepase su capacidad de carga *W*.

Objetivo

- Llenar la mochila de tal forma que se maximice el valor de los objetos transportados en ella.
- Hay que tener en cuenta que la suma de los pesos de los objetos seleccionados no debe superar la capacidad máxima *W* de la mochila.

Problema de la mochila – Planteamiento matemático

- Maximizar: $\sum_{i=1}^{n} x_i v_i$
- Sujeto a la restricción: $\sum_{i=1}^{n} x_i w_i \leq M$
- Tipos de Mochila:
  - La mochila fraccionaria (siempre funciona la estrategia voraz o greedy).
  - La mochila entera (0/1) (no siempre funciona la estrategia voraz).
- Se trata de resolver el problema de la mochila binaria 0/1 mediante búsqueda exhaustiva.

Problema de la mochila – Estrategia de solución

- La búsqueda estará basada en el siguiente árbol:
  - En cada nivel se puede decidir si elegir o no el artículo.
  - La cantidad de decisiones que se deben tomar depende de la cantidad de artículos y de no sobrepasar la capacidad de carga de la mochila.
  - Se debe llenar la mochila con aquellos objetos que maximicen el valor que lleva.

Sudoku

- El Sudoku se basa en el trabajo sobre *Cuadrados Latinos* de Leonard Euler.
- Un *cuadrado latino* es una matriz de $n_x n$ elementos en la que cada celda contiene uno de los n símbolos de tal modo que cada uno de ellos aparece exactamente una vez en cada columna y en cada fila. Se llaman así porque Euler utilizó caracteres latinos para representarlo.

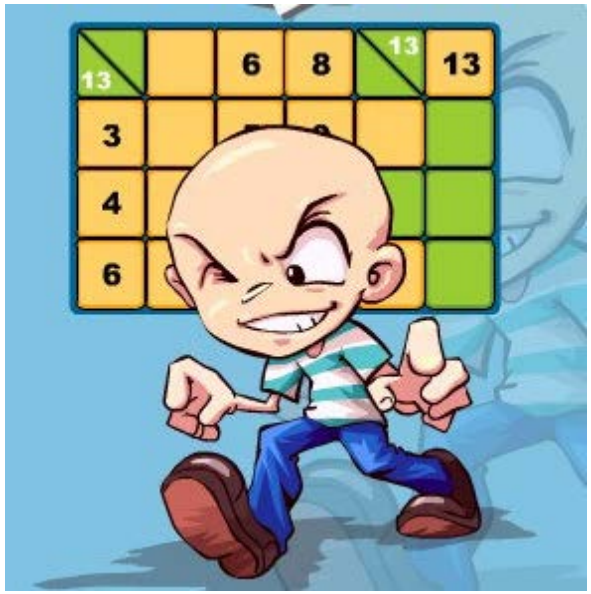
1	2	3
2	3	1
3	1	2

a	b	d	c
b	c	a	d
c	d	d	a
d	a	c	b

Sudoku

- Sudoku: **su** = número, **doku** = solo
- Es un tablero de 9x9 compuesto por regiones cuadradas de 3x3. Algunas celdas ya tienen números, el objetivo es rellenar las celdas vacías, con un número de tal forma que cada columna, fila y región contengan los números del 1 al 9 sin que se repitan.
- El SUDOKU añade una restricción adicional a los cuadrados latinos los subgrupos de 3x3 deben tener los dígitos del 1 al 9.
- Un Sudoku bien planteado debe tener una única solución.

Sudoku



5	2		3		4		6	
1	8	4	6		9		3	
	6	9		7	8	4		
7	4			3		1		5
		1	2	9	7		4	
9	3	8		4	5		7	
		5	9				8	
4	9			8	1	2	5	
			4					3

Laberinto - Plantemiento

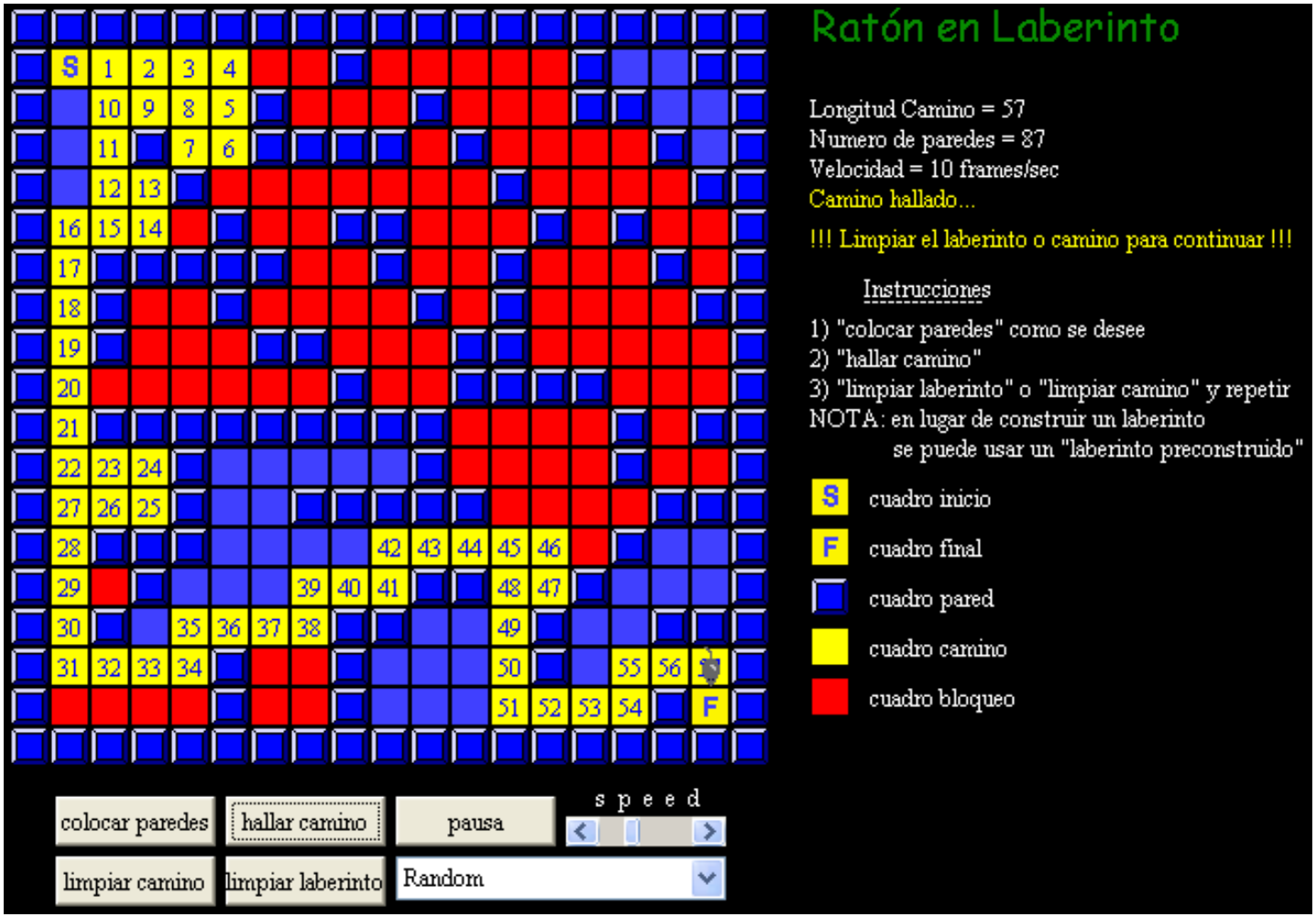
- El problema consiste en hallar una ruta de salida de un laberinto.
- El laberinto se representa mediante un array (dimensión máxima de 30x30), cuyo contenido serán constantes enteras (1 celda ocupada, 0 si no lo está).
- Los datos del laberinto se leen desde un fichero de tipo texto (laberinto.dat) que contiene los siguientes datos:
  - Número de filas y columnas del laberinto.
  - Fila y columna de la posición inicial.
  - Fila y columna de la posición final.
  - Por cada posición ocupada la fila y columna.

  Nota: La numeración de las filas y coumnas es a partir de 1 (para el usuario).
- Los movimientos permitidos son avanzar o retroceder una celda en la fila o columna, es decir cuatro movimientos como máximo. No se consideran movimientos en diagonal.

Laberinto - Algoritmo

- El algoritmo a seguir para resolver el programa es el siguiente:
  - **Asignar** como posiciones ocupadas: la primera y posterior a la última fila, así como la primera y posterior a la última columna (para facilitar la programación).
  - **Inicializar** el contador de movimientos, el contador de posiciones del camino y el camino (array).
  - **Mientras** no sea la posición final realizar:
    - **Mientras** no se supere el límite de movimientos (4) comprobar, siguiendo un orden (antihorario), si la casilla siguiente está desocupada. Si es así salir del bucle.
    - **Si** el contador de movimientos está en el rango correcto, **incrementar** el contador de posiciones del camino, **almacenar** la fila y columna respectiva en el array del camino y **guardar** el contador de movimiento (en un array), **marcar** como ocupada la fila y columna respectiva y **reinicializar** el contador de movimientos. **Sino**, **si** el contador de posiciones del camino es cero, devolver un error (no hay camino), **caso contrario**, volver a la posición anterior **disminuyendo** en uno el contador de posiciones del camino, **recuperando** la fila y columna respectiva a esa posición y **recuperando** el número del movimiento (para continuar probando las demás posiciones).
  - **Almacenar** la última posición del camino.

Laberinto


Ratón en Laberinto
Longitud Camino = 57
Numero de paredes = 87
Velocidad = 10 frames/sec
Camino hallado...
!!! Limpiar el laberinto o camino para continuar !!!
Instrucciones
1) "colocar paredes" como se desee
2) "hallar camino"
3) "limpiar laberinto" o "limpiar camino" y repetir
NOTA: en lugar de construir un laberinto
se puede usar un "laberinto preconstruido"
S cuadro inicio
F cuadro final
cuadro pared
cuadro camino
cuadro bloqueo
colocar paredes
hallar camino
pausa
speed
limpiar camino
limpiar laberinto
Random

Preprocesador y programas multifichero

Índice

- Directrices del preprocesador.
- Constantes y Macros.
- Inclusión de ficheros.
- Sentencias Condicionales.

Directivas del preprocesador

- Son expandidas en la fase de preprocesado:
  - **`#define`** : Define una nueva constante o macro del preprocesador.
  - **`#include`** : Incluye el contenido de otro fichero.
  - **`#ifdef #ifndef`** : Preprocesamiento condicionado.
  - **`#endif`** : Fin de bloque condicional.
  - **`#error`** : Muestra un mensaje de error

Constantes y Macros

- Permite asociar valores constantes a ciertos identificadores expandidos en fase de preprocesamiento:

```
#define variable valor
```

- Define funciones que son expandidas en fase de preprocesamiento:

```
#define macro(args,...) función
```

Constantes y Macros. Ejemplos

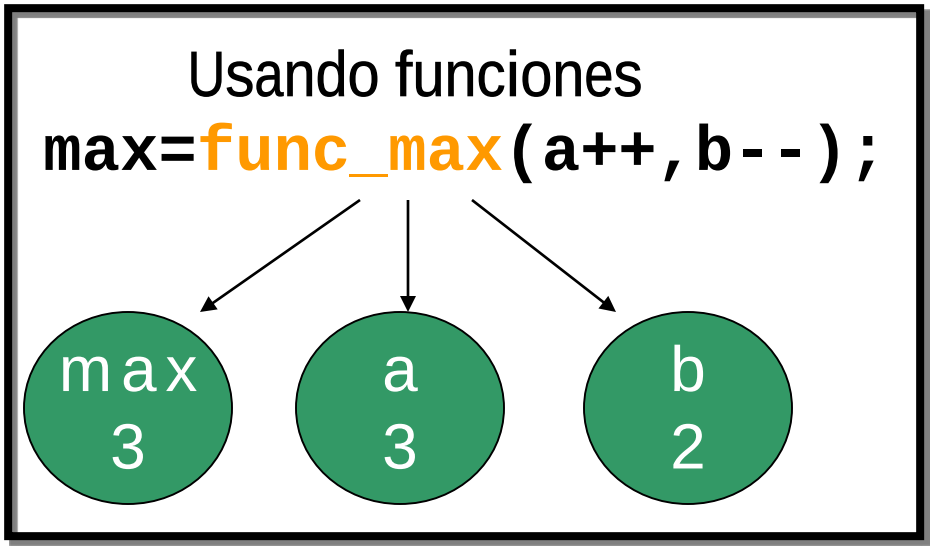
```
#define PI          3.14
#define NUM_ELEM    5
#define AREA(rad)   PI*rad*rad
#define MAX(a,b)    (a>b ? a : b)
int main()
{
  int   i;
  float vec[NUM_ELEM];
  for(i=0;i<NUM_ELEM;i++)
     vec[i]=MAX((float)i*5.2,AREA(i));
}
```

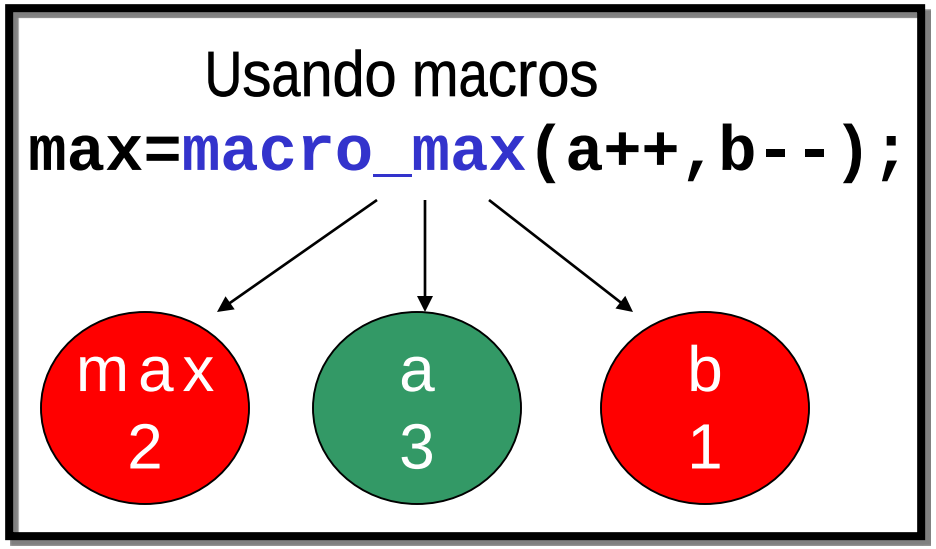
Constantes y Macros. Después del preprocesamiento

```
int main()
{
  int   i;
  float vec[5];
  for(i=0;i<5;i++)
     vec[i]=((float)i*5.2>3.14*i*i ?
             (float)i*5.2 :
             3.14*i*i);
}
```

Macros vs Funciones

```
#define macro_max(a,b) (a>b ? a : b)
int func_max(int a, int b)
{ return (a>b ? a : b); }
int a=2,b=3,max;
```

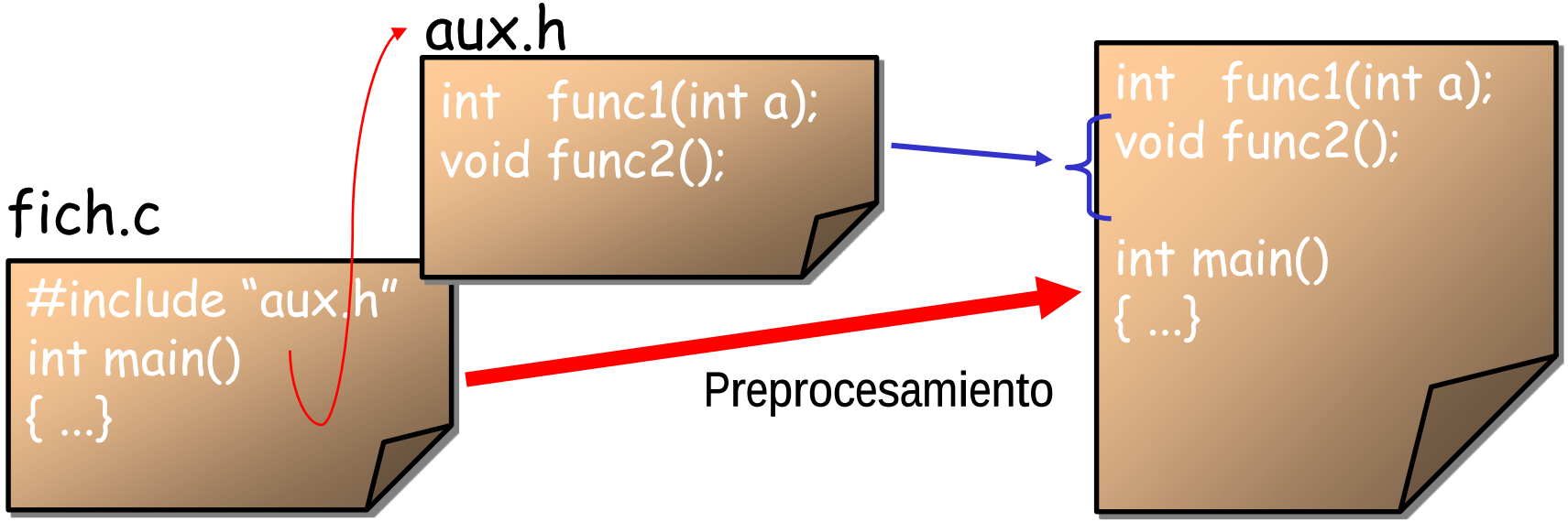




Inclusión de Ficheros

- Los prototipos de las funciones usadas por varios ficheros fuente se suelen definir en fichero de cabecera.

```
#include <stdio.h>          Cabeceras del sistema
#include "mis_func.h"       Ficheros de cabecera locales
```



Inclusión de Ficheros

- La inclusión de ficheros esta sujeta a las siguientes recomendaciones:
  - Por lo general los ficheros de cabecera tienen como extensión .h
  - En los ficheros de cabecera no se incluyen implementación de funciones
  - Las variables en un fichero de cabecera son declaradas extern y se encuentran declaradas en algún otro fichero .c

Sentencias Condicionales

- Para incluir código cuya compilación es dependiente de ciertas opciones, se usan los bloques:

```
#ifdef variable
<bloque de sentencias>
...
#endif
#ifndef variable
<bloque de sentencias>
...
#endif
```

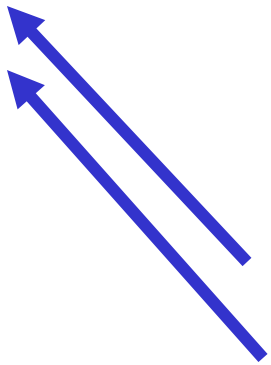
Ejemplo: Depuración

```
#define DEBUG
int main()
{
  int i,acc;
  for(i=0;i<10;i++)
     acc=i*i-1;
#ifdef DEBUG
  printf("Fin bucle acumulador: %d",acc);
#endif
  return 0;
}
```

Ejemplo: Fichero de cabecera

aux.h

```
#ifndef _AUX_H_
#define _AUX_H_
<definiciones>
#endif
```



Evita la redefinicion de funciones y variables

```
#include “aux.h”
#include “aux.h”
int main()
{
...
}
```

Compilación en modo comando

- Visual C se puede usar desde la línea de comandos:

```
> cl –c pp.c
  para compilar fuentes
> cl –o pp.exe pp.obj  pp1.obj
  para enlazar objetos
```

Simulación dinámica en C

Índice General

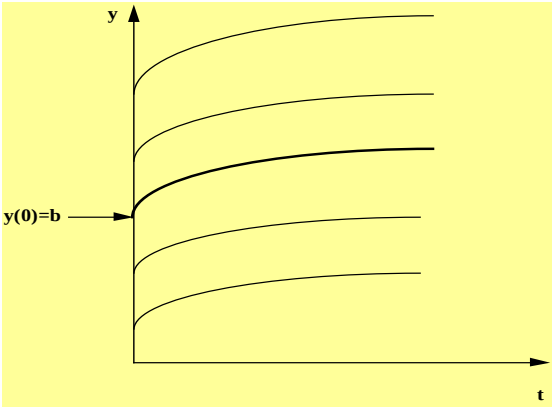
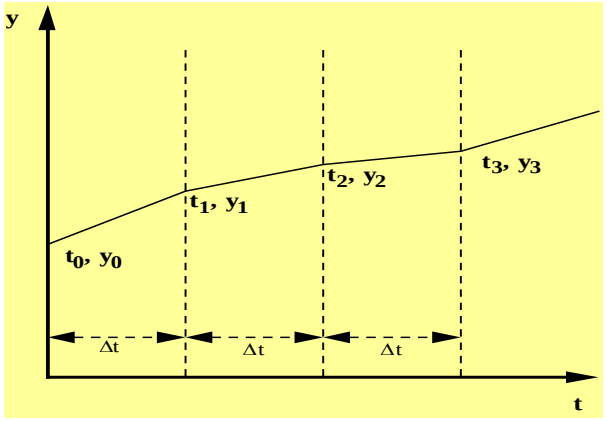
Introducción

- La dinámica de los sistemas responden a leyes físicas que se expresan habitualmente en forma de **ecuaciones diferenciales**.
- Ejemplo: ecuaciones del movimiento de los cuerpos (segunda ley de Newton) → ecuación diferencial de segundo orden
- Pocas ecuaciones diferenciales tienen una solución analítica sencilla, así que la solución pasa por obtener una solución aproximada mediante la aplicación de *métodos numéricos*.

Ecuaciones diferenciales

- *Ecuación Diferencial*: Ecuaciones que involucran variables dependientes y sus derivadas con respecto a las variables independientes.
- *Ecuación Diferencial Ordinaria*: ecuaciones diferenciales que involucran solamente UNA variable independiente.
- *Ecuación Diferencial Parcial*: : ecuaciones diferenciales que involucran dos o más variables independientes.

Solución de EDOs: Analítica y Numérica

Método de Solución Analítica	Método de Solución Numérica
	
▪Resolver la EDO para encontrar una familia de soluciones. ▪Elije la solución que satisface las condiciones iniciales correctas. Encuentra una fórmula analítica parar *y(t)*	▪Empieza con las condic. iniciales ▪Resuelve para pequeños tamaños de paso (Δt). ▪Resuelve aprox. en cada Δt ▪Encuentra pares de puntos: (t_0,y_0), (t_1,y_1), …

Solución de EDOs mediante métodos de un solo paso

- El objetivo consiste en obtener un nuevo punto a partir de un punto anterior (solución en forma discreta)
- Se basan en el método de Taylor de orden k:
  - Sea una EDO de primer orden: $y' = f(x, y)$

    $y(x_0) = y_0$

    $x \in [a, b]$
  - Se aplica la serie de Taylor para aproximar la solución de la EDO, haciendo: $y'(x_i) = y'_i$

Solución de EDOs mediante el método de Taylor de orden k

- Se plantea el siguiente algoritmo:

$$Dados: x_0, y_0, h, n$$

$$Para\ i = 1, 2, 3, \cdots n$$

$$x_{i+1} = x_i + h$$

$$y_{i+1} = y_i + h\,y_i' + \frac{h^2}{2!}y_i'' + \frac{h^3}{3!}y_i''' + \cdots + \frac{h^k}{k!}y_i^{(k)}$$

- Siendo E el error de truncamiento:

$$E = \frac{h^{k+1}}{(k+1)!}y^{(k+1)}(\xi) \quad x_i < \xi < x_{i+1}$$

Método de Euler

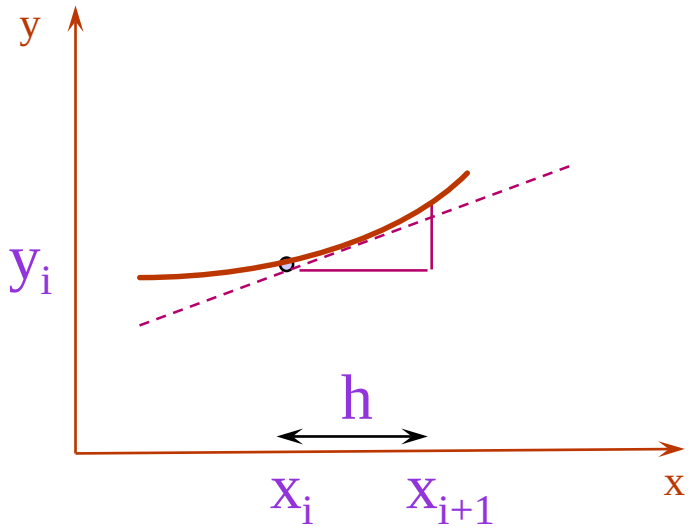
- Permite resolver una EDO de primer orden de la forma: $\frac{dy}{dt} = f(t, y)$

  $y(t_0) = y_0$

- Algoritmo:

$$Dado: (t_0, y_0), h \; y \; n$$
$$Para \; i = 0, 1, 2, \cdots n$$
$$t_{i+1} = t_i + h$$
$$y_{i+1} = y_i + hf(t_i, y_i)$$

Interpretación geométrica



Método de Euler: características

- Sencillo. La primera derivada proporciona una estimación de la pendiente en x_i
- La ecuación se aplica iterativamente, con un paso pequeño para reducir el error
- Análisis del error:
  - *Error de truncamiento* - causado por la naturaleza de la técnica empleada para aproximar los valores de y
    - error global = error local de truncamiento (a partir de la Serie de Taylor) + propagación del error de truncamiento
  - *Error de Redondeo* – causado por el número limitado de dígitos significativos que retiene el ordenador

Método de Euler: características

- Un error fundamental del método de Euler es que se aplica la derivada asumida en el principio del intervalo en todo el intervalo.
- Si h disminuye aumenta la precisión a costa de la complejidad (tiempo) del cálculo que crece.
- Si h disminuye demasiado la precisión se ve afectada debido a la acumulación de errores de redondeo.
- Con una simple modificación en el cálculo de la derivada, se puede obtener mejores resultados (RK).

Ejemplo de aplicación del método de Euler

- Resolver la siguiente EDO:

$$\frac{dy(t)}{dt} + 2y(t) = 3e^{-4t}$$

$$y(0) = 1$$

$$t_0 = 0, \quad t_{\max} = 4, \quad h = 0.1$$

- Para comparar la solución exacta es: $y(t) = \frac{5e^{-2t} - 3e^{-4t}}{2}$

Programa C que resuelve el ejemplo Euler

```
/*****************************************************************\
* Programa: euler.c                                               *
* Descripción: Resuleve una ecuacion diferencial de primer orden usando  *
*              el metodo de Euler                                 *
* Ecuacion a resolver: y'(t)=3*exp(-4*t)-2*y(t)   con y(0) = 1    *
* La solucion exacta de esta ecuacion es:                         *
*                y(t)=2.5*exp(-2*t)-1.5*exp(-4*t)                 *
* El programa se puede adaptar para resolver cualquier ecuacion EDO de   *
* primer orden modificando la funcion derivada que calcula la     *
* derivada y el valor de NDERIVS                                  *
* Autor: Pedro Corcuera                                           *
\*****************************************************************/
#include <stdio.h>          // I/O library
#include <math.h>           // Math library
#define  NDERIV   1         // numero de derivadas

void derivada(int numDerivs, float *x, float t, float *k1);
void main(void);

// derivada calcula el numero de derivadas
// x array que contiene las variables de estado
// t tiempo
// k1 array que contiene en valor de las derivadas calculadas
void derivada(int numDerivs, float *x, float t, float *k1) {
   k1[0]=3*exp(-4*t)-2*x[0];        // calcula derivada
}
```

Programa C que resuelve el ejemplo Euler

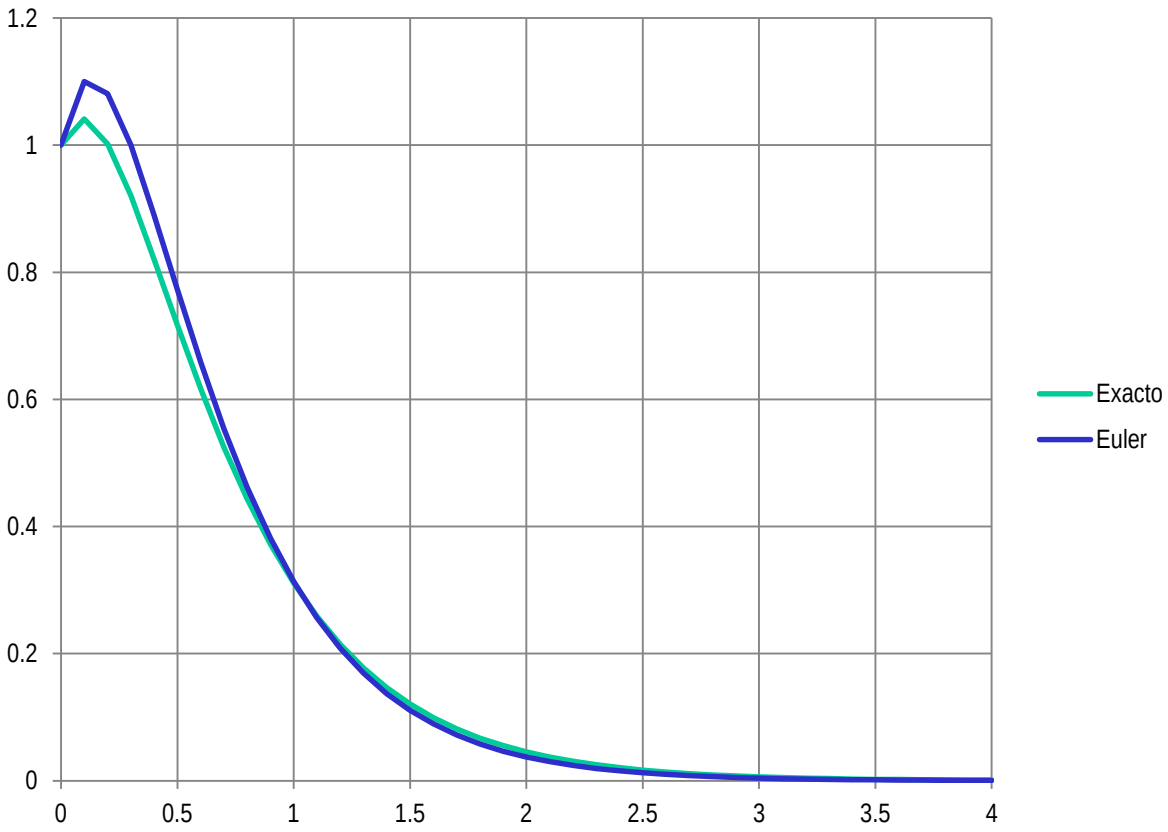
```
void main(void) {
   float yexact, yaprox;        // Valores exactos y aproximados de y
   float x[NDERIV], k1[NDERIV];// arrays almacena estado de las variables y derivadas
   float h, t, t0, tmax, y0;   // paso,tiempo,tiempo inicial,tiempo final,y inicial
   int i;                       // indice de contador

    // Lectura de intervalo de calculo
    printf("Ingresa t inicial: ");
    scanf(" %f", &t0);
    printf("Ingresa t max: ");
    scanf(" %f", &tmax);

    // Lectura de condicion inicial
    printf("Ingresa condicion inicial: ");
    scanf(" %f", &y0);

    // Lectura de paso h
    printf("Ingresa paso h: ");
    scanf(" %f", &h);

   x[0]=y0;                          // condiciones iniciales
   yaprox=x[0];                      // resultado inicial

   printf("Solucion de ODE mediante Euler\n");
   printf("   t \t yexacto \tyaprox\n");
```

Programa C que resuelve el ejemplo Euler

```
    for (t = t0; t < tmax; t+=h) {    //ciclo for para iterar sobre el tiempo
          derivada(NDERIV, x, t, k1);

          yaprox=x[0];                // obtiene valor aproximado y exacto de y
          yexact=2.5*exp(-2*t)-1.5*exp(-4*t);

          // Imprime resultados
          printf(" %4.2f \t %6.4f \t%6.4f\n", t, yexact, yaprox);
         // actualiza variables de estado
          for(i = 0; i < NDERIV; i++)
                    x[i] = x[i] + h*k1[i];
    }
    printf("Programa finalizado\n");
}
```

Resultados del programa ejemplo (Excel)

t	yexacto	yaprox
0	1	1
0.1	1.0413	1.1
0.2	1.0018	1.0811
0.3	0.9202	0.9997
0.4	0.8205	0.8901
0.5	0.7167	0.7726
0.6	0.6169	0.6587
0.7	0.5253	0.5542
0.8	0.4436	0.4616
0.9	0.3723	0.3815
1	0.3109	0.3134
1.1	0.2586	0.2562
1.2	0.2145	0.2087
1.3	0.1774	0.1694
1.4	0.1465	0.1372
1.5	0.1207	0.1108
1.6	0.0994	0.0894
1.7	0.0818	0.072
1.8	0.0672	0.058
1.9	0.0552	0.0466
2	0.0453	0.0374
2.1	0.0372	0.03
2.2	0.0305	0.0241
2.3	0.025	0.0193
2.4	0.0205	0.0155
2.5	0.0168	0.0124
2.6	0.0137	0.0099
2.7	0.0113	0.008
2.8	0.0092	0.0064
2.9	0.0076	0.0051
3	0.0062	0.0041
3.1	0.0051	0.0033
3.2	0.0041	0.0026
3.3	0.0034	0.0021
3.4	0.0028	0.0017
3.5	0.0023	0.0013
3.6	0.0019	0.0011
3.7	0.0015	0.0009
3.8	0.0013	0.0007
3.9	0.001	0.0006
4	0.0008	0.0004



Experimentos con el método Euler

- Disminuir paso h para obtener mayor precisión y comprobar el aumento del tiempo requerido para el cálculo.
- Comprobar que si h disminuye demasiado, la precisión se ve afectada debido a la acumulación de errores de redondeo.

Método de Euler en EDOs de mayor orden

- El *truco* es descomponer la ecuación diferencial de mayor orden en varias EDOs de primer orden. A esta técnica se le llama análisis de variable de estado
- Ejemplo: $\frac{d^2y(t)}{dt^2} + 100\frac{dy(t)}{dt} + 10^4 y(t) = 10^4\left|\sin(377t)\right|$

$$Condiciones\ iniciales\ cero: y(0) = 0, y'(0) = 0$$

$$t_0 = 0, \quad t_{\max} = 0.08, \quad h = 0.0001$$

- corresponde al circuito rectificador del libro Circuits Devices and Systems, de Smith y Dorf (sección 7.3) con parámetros: R=400W, C=25mF, L=4H, w=377 rad/sec(60 Hz), V_m=1. El valor absoluto del seno equivale a un rectificador y y(t) es el voltaje de salida

Método de Euler en EDOs de mayor orden

- Reducción a varias EDOs de primer orden:

$$Introducimos\ x_1(t),\ x_2(t).\ Hacemos:$$

$$x_1(t) = y(t)$$

$$\frac{dx_1(t)}{dt} = x_2(t)$$

$$\frac{dx_2(t)}{dt} = -100\frac{dy(t)}{dt} - 10^4 y(t) + 10^4\left|\sin(377t)\right|$$

$$= -10^4 x_1(t) - 100x_2(t) + 10^4\left|\sin(377t)\right|$$

Método de Euler en EDOs de mayor orden

- En términos de ec. matricial de primer orden:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -10^4 & -100 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 10^4 \end{bmatrix} \left|\sin(377t)\right|$$

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\left|\sin(377t)\right|$$

$$donde: \mathbf{A} = \begin{bmatrix} 0 & 1 \\ -10^4 & -100 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 \\ 10^4 \end{bmatrix}$$

Solución mediante método Euler

- Para resolver ambas ecuaciones de primer orden simultáneamente se aplica el siguiente algoritmo:
  1. Empezar en tiempo t_0, con un valor para h y condiciones iniciales de las variables de estado $x_1(t_0)$, $x_2(t_0)$
  2. Con los valores $x_i(t_0)$ calcular las derivadas para cada $x_i(t)$ en $t= t_0$, llamándoles k_{1i}
  3. Calcular el valor aproximado de cada $x^*_i(t_0+h)=x^*_i(t_0)+ k_{1i} h$
  4. Hacer $t_0= t_0+h$ y para cada $x_i(t_0)=x^*_i(t_0+h)$
  5. Repetir pasos 2 a 4 hasta alcanzar el tiempo de simulación

Programa C que resuelve EDO de 2 orden

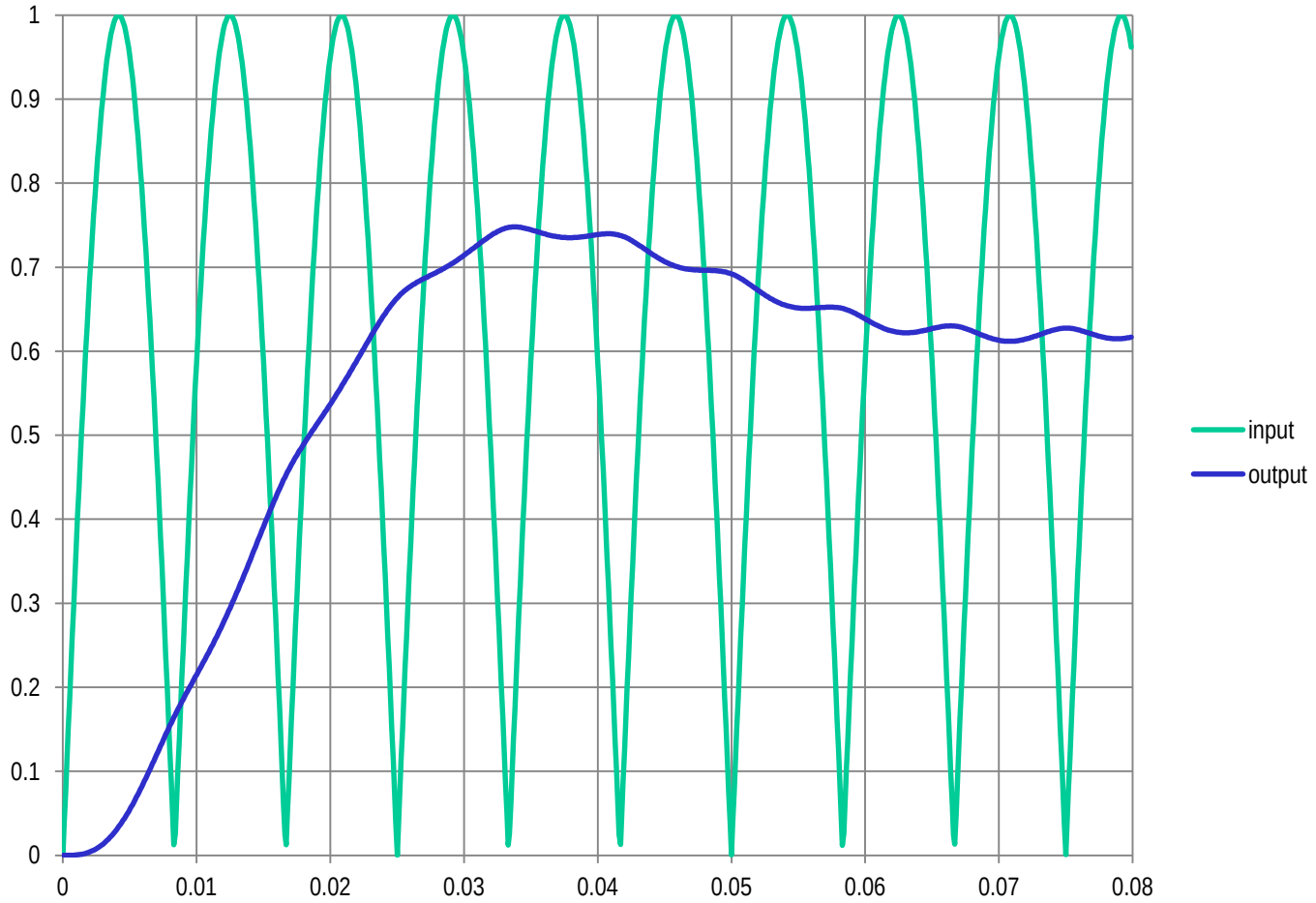
```
/*****************************************************************\
* Programa: euler2.c                                              *
* Descripción: Resuleve una ecuacion diferencial de segundo orden usando *
*              el metodo de Euler                                 *
* Ecuacion a resolver: y''(t)+100*y'(t)+1E4*y(t)=1E4*abs(sin(377t))      *
*                      con y(0) = 0, y'(0)=1                      *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 4/02/2008                                         *
\*****************************************************************/

#include <stdio.h>          // I/O library
#include <math.h>           // Math library

#define  NDERIV   2                 // numero de derivadas

void derivada(int numDerivs, float *x, float t, float *k1);
void main(void);

// derivada calcula el numero de derivadas
// x array que contiene las variables de estado
// t tiempo
// k1 array que contiene en valor de las derivadas calculadas
void derivada(int numDerivs, float *x, float t, float *k1) {
   k1[0]=x[1];        // calcula derivada
   k1[1]=1.0E4*fabs(sin(377.0*t))-100*x[1]-1.0E4*x[0];
}
```

Programa C que resuelve EDO de 2 orden

```
 void main(void) {
   float yaprox;                // Valores aproximados de y
   float x[NDERIV], k1[NDERIV];// arrays almacena estado de variables y derivadas
   float h=0.0001, t=0.0, tmax=0.08; // paso, tiempo, tiempo inicial, tiempo final
   int i;                       // indice de contador

   x[0]=0.0;                                    // condicionles iniciales
   x[1]=0.0;

   printf("Solucion de ODE mediante Euler\n");
   printf("   t(ms) \t input \t output\n");

   for (t = 0; t < tmax; t+=h) {              //ciclo for para iterar sobre el tiempo
         derivada(NDERIV, x, t, k1);
         yaprox=x[0];                                  // obtiene valor de y
         // Imprime resultados
         printf(" %7.4f \t %7.4f \t%7.4f\n", t, fabs(sin(377.0*t)), yaprox);
        // actualiza variables de estado
         for(i=0;i<NDERIV;i++)
                  x[i] = x[i] + h*k1[i];
   }
   printf("Programa finalizado\n");
}
```

Resultados del programa euler2 (Excel)

t(ms)	input	output
0	0	0
0.0001	0.0377	0
0.0002	0.0753	0
0.0003	0.1129	0
0.0004	0.1502	0
0.0005	0.1874	0
0.0006	0.2243	0.0001
0.0007	0.2608	0.0001
0.0008	0.297	0.0002
0.0009	0.3328	0.0003
0.001	0.3681	0.0004
0.0011	0.4029	0.0006
0.0012	0.4371	0.0008
0.0013	0.4707	0.001
0.0014	0.5036	0.0013
0.0015	0.5358	0.0016
0.0016	0.5673	0.002
0.0017	0.5979	0.0024
0.0018	0.6277	0.0029
0.0019	0.6566	0.0034
0.002	0.6846	0.004
...		
...		



Métodos de Runge-Kutta

- Todos los métodos de Runge-Kutta son generalizaciones de la fórmula básica de Euler, en la que la función pendiente *f* se remplaza por un promedio ponderado de pendientes en el intervalo $x_n \leq x \leq x_{n+1}$

$$y_{n+1} = y_n + h(w_1 k_1 + w_2 k_2 + \cdots + w_m k_m)$$

donde las ponderaciones w_i, $i = 1, 2, \ldots, m$ son constantes que satisfacen $w_1 + w_2 + \ldots + w_m = 1$, y k_i es la función evaluada en un punto seleccionado (x, y) para el cual $x_n \leq x \leq x_{n+1}$.

Métodos de Runge-Kutta

- El número *m* se llama el ***orden***. Si $m = 1$, $w_1 = 1$, $k_1 = f(x, y_n)$, equivale al método de *Euler*, que es un ***método de Runge-Kutta de primer orden***.
- Método de RK de orden 2: (Heun)

$$\frac{dy}{dx} = f(x, y)$$
$$y(x_0) = y_0$$

$$Dado \quad (x_0, y_0)\ y\ h$$
$$Para\ n = 0, 1, 2, \cdots$$
$$x_{n+1} = x_n + h$$
$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf(x_n + h, y_n + k_1)$$
$$y_{n+1} = y_n + \frac{k_1 + k_2}{2}$$

Métodos de Runge-Kutta

- Método de RK de orden 4:
  (muy empleado)

$$\frac{dy}{dx} = f(x, y)$$
$$y(x_0) = y_0$$

$$Dado \quad (x_0, y_0)\ y\ h$$
$$Para\ n = 0, 1, 2, \cdots$$
$$x_{n+1} = x_n + h$$
$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$
$$k_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right)$$
$$k_4 = hf(x_n + h, y_n + k_3)$$
$$y_{n+1} = y_n + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}$$

Programa C que resuelve el ejemplo RK4

```
/*****************************************************************\
* Programa: rk4_ejem.c                                            *
* Descripción: Resuleve una ecuacion diferencial de primer orden usando  *
*              el metodo de Runge-Kutta  4 orden                  *
* Ecuacion a resolver: dy/dt = exp(-2*t) - 3*y(t)  con y(0) = 1   *
* Solucion para comprobar: y(t) = exp(-2*t)                       *
* El resultado lo imprime en el fichero outputrk.txt              *
\*****************************************************************/
#include <stdio.h>
#include <math.h>

#define N         1       /* numero de ec. de primer orden */
#define H     0.005       /* paso de tiempo t*/
#define MAX   1.001       /* valor max de t */
FILE *output;              /* nombre fichero interno */
double runge4(double x, double y, double step);   /* funcion Runge-Kutta */
double f(double x, double y);            /* funcion para las derivadass */

int main(void) {
  double t, y, exactf,perc_diff;   int j;

  output=fopen("outputrk.txt", "w");            /* nombre de fichero externo */
  t = 0.0;  y = 1.0;                            /* valor inicial */
  exactf = exp (-2.0*t);

  fprintf(output, "%6.3f\t%f\t%f\n", t, y,exactf);
```

Programa C que resuelve el ejemplo RK4

```
  for (j=1; j*H <= MAX ;j++) {                    /* ciclo del tiempo */
       t=j*H;
       y = runge4(t, y, H);
       exactf = exp (-2.0*t);
       perc_diff = (exactf-y)*100.0/exactf;
       fprintf(output, "%6.3f\t%f\t%lf\t%lf\n", t, y,exactf,perc_diff);
  }
  fclose(output);  printf("Programa finalizado\n");  return 0;
}

double runge4(double x, double y, double step){
  double h=step/2.0,             /* mitad de paso */
  t1, t2, t3, k1, k2, k3,k4;     /* variables temporales para Runge-Kutta */
  int i;

  t1= y +0.5*(k1=step*f(x, y));
  t2= y +0.5*(k2=step*f(x+h, t1));
  t3= y + (k3=step*f(x+h, t2));
  k4= step*f(x+step, t3);
  y +=(k1+2*k2+2*k3+k4)/6.0;
  return y;
}

double f(double x, double y){
  return(exp(-2*x) - 3*y);     /* derivada de la primera ecuacion */
}
```

Resultados del programa RK4 (fichero outputrk.txt)

```
0.000   1.000000 1.000000
0.005   0.990001 0.990050 0.004963
0.010   0.980102 0.980199 0.009901
0.015   0.970302 0.970446 0.014814
0.020   0.960600 0.960789 0.019703
0.025   0.950996 0.951229 0.024567
0.030   0.941488 0.941765 0.029407
0.035   0.932075 0.932394 0.034223
0.040   0.922756 0.923116 0.039015
0.045   0.913531 0.913931 0.043783
0.050   0.904398 0.904837 0.048528
0.055   0.895357 0.895834 0.053248
0.060   0.886407 0.886920 0.057945
0.065   0.877546 0.878095 0.062619
0.070   0.868773 0.869358 0.067269
0.075   0.860089 0.860708 0.071896
0.080   0.851492 0.852144 0.076501
0.085   0.842981 0.843665 0.081082
0.090   0.834555 0.835270 0.085640
0.095   0.826213 0.826959 0.090175
0.100   0.817956 0.818731 0.094688
...
0.990   0.137206 0.138069 0.625292
0.995   0.135838 0.136695 0.627136
1.000   0.134484 0.135335 0.628970
```

Ejemplo de utilización de la rutina rk4.c de la librería Numerical Recipes

- El libro Numerical Recipes in C contiene una colección de rutinas muy elaboradas en C para cualquier tipo de problema numérico.
- Conviene saber compilar y utilizar tales rutinas:
  - En CodeBlocks se crea un proyecto y se añaden los ficheros necesarios
    - rk4.c
    - nrutil.c, bessj.c, bessj0.c, bessj1.c
    - nr.h, nrutil.h
    - driver (rutina que resuelve el problema ej_ej_rk4.c)

Rutina RK4 de Numerical Recipes

```
#include "nrutil.h"
void rk4(float y[], float dydx[], int n, float x, float h, float yout[],
    void (*derivs)(float, float [], float []))
{
    int i;
    float xh,hh,h6,*dym,*dyt,*yt;

    dym=vector(1,n); dyt=vector(1,n); yt=vector(1,n);
    hh=h*0.5;
    h6=h/6.0;
    xh=x+hh;
    for (i=1;i<=n;i++) yt[i]=y[i]+hh*dydx[i];
    (*derivs)(xh,yt,dyt);
    for (i=1;i<=n;i++) yt[i]=y[i]+hh*dyt[i];
    (*derivs)(xh,yt,dym);
    for (i=1;i<=n;i++) {
          yt[i]=y[i]+h*dym[i];
          dym[i] += dyt[i];
    }
    (*derivs)(x+h,yt,dyt);
    for (i=1;i<=n;i++)
          yout[i]=y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);
    free_vector(yt,1,n);
    free_vector(dyt,1,n);
    free_vector(dym,1,n);
}
```

puntero a función

Uso de la rutina RK4 de Numerical Recipes

```
#include <stdio.h>
#include "nr.h"
#include "nrutil.h"
#define N 4
void derivs(float x,float y[],float dydx[]) {
    dydx[1] = -y[2]; dydx[2]=y[1]-(1.0/x)*y[2];
    dydx[3]=y[2]-(2.0/x)*y[3]; dydx[4]=y[3]-(3.0/x)*y[4];
}
int main(void) {
    int i,j;
    float h,x=1.0,*y,*dydx,*yout;
    y=vector(1,N); dydx=vector(1,N); yout=vector(1,N);
    y[1]=bessj0(x); y[2]=bessj1(x); y[3]=bessj(2,x); y[4]=bessj(3,x);
    derivs(x,y,dydx);
    printf("\n%16s %5s %12s %12s %12s\n", "Bessel function:","j0","j1","j3","j4");
    for (i=1;i<=5;i++) {
          h=0.2*i;
          rk4(y,dydx,N,x,h,yout,derivs);
          printf("\nfor a step size of: %6.2f\n",h); printf("%12s","rk4:");
          for (j=1;j<=4;j++) printf(" %12.6f",yout[j]);
          printf("\n%12s %12.6f %12.6f %12.6f %12.6f\n","actual:",
                   bessj0(x+h),bessj1(x+h),bessj(2,x+h),bessj(3,x+h));
    }
    free_vector(yout,1,N); free_vector(dydx,1,N); free_vector(y,1,N);
    return 0;
}
```

Medida del tiempo y sleep

- Se puede utilizar las funciones **`time`** y **`clock`** clock que se encuentran en <time.h> para medir el tiempo de ejecución en segundos y milisegundos de un conjunto de instrucciones
- Otra función que se puede emplear para hacer programas que se ejecutan en tiempo real es **`_sleep(t)`** donde t es el tiempo en milisegundos que se desea no haga nada el programa

Ejemplo de medida del tiempo y sleep

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

int main(void)
{   long i,j;
    double y=0;
    time_t start,end;
    double dif;
    int t,t0;

     t0 = clock();
    for (i=0;i<10;i++) {
       time (&start);
       y=0;
       for (j=0;j<=1000000;j++) y=y+5*j+j*i;
       _sleep(1000); // tiempo en milisegundos
       time (&end);
       dif = difftime (end,start);
       printf ("Tiempo: %lf seconds ", dif );
   }
   t = clock();
 printf ("Toma %d clicks (%f sec). \n",t, (float)(t-t0)/CLOCKS_PER_SEC);
   return 0;
}
```

Simulación de un control PID

- Los controladores PID son muy utilizados en la industria por su sencillez.
- Es un control con términos Propocional, Integral y Derivativo que actúa sobre una señal de error
- Ecuación standard:

$$u(t) = K\left(e(t) + \frac{1}{T_i}\int_0^t e(t)dt + T_d \frac{d}{dt}e(t) \right)$$

donde: *u(t)* es la salida del controlador
e(t) entrada al control, diferencia entre el valor de consigna o tarado (setpoint) y la salida de la planta
K ganancia proporcional, T_i tiempo integral, T_d tiempo de derivación

Simulación de un control PID

- Forma paralela de la ecuación:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{d}{dt} e(t)$$

  - se puede convertir los parámetros de una forma a otra:

$$K_p = K$$

$$K_i = \frac{K}{T_i}$$

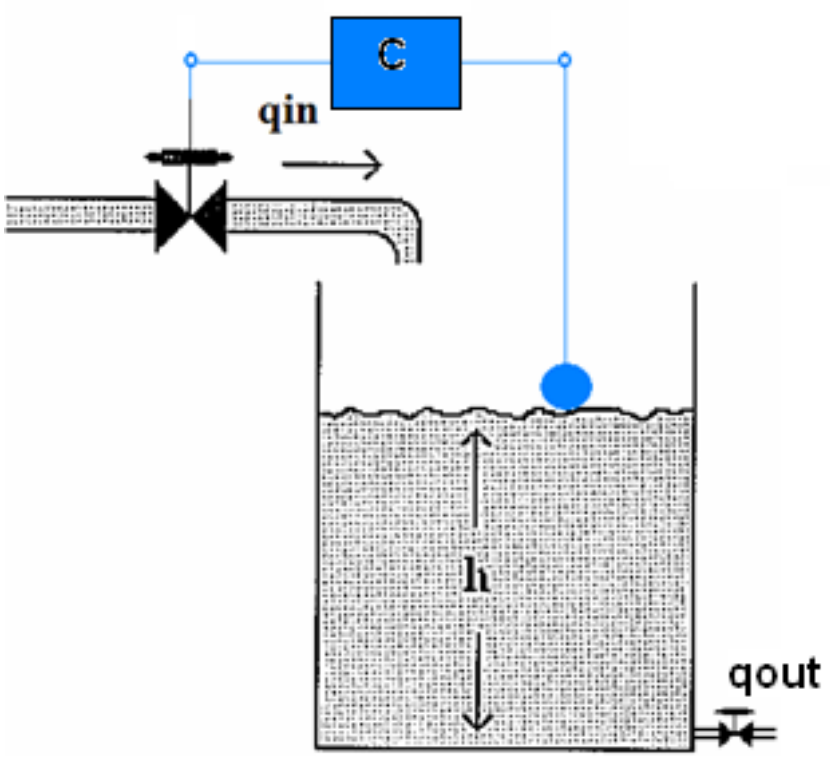
$$K_d = K T_d$$

Código de control PID

```
#include <math.h>
//Parametros
#define epsilon 0.001
#define MAX  1                    //para saturación
#define MIN -1
#define Kp  1.0     // 0.1
#define Kd  0.01    // 0.01
#define Ki  0.1     // 0.005

float PIDcal(float setpoint,float actual_position, float dt) {
    static float pre_error = 0; static float integral = 0;
    float error,derivative,float output;

    //Calculo P,I,D
    error = setpoint - actual_position;
    //si el error es muy pequeño no se integra
    if(fabs(error) > epsilon) integral = integral + error*dt;
    derivative = (error - pre_error)/dt;
    output = Kp*error + Ki*integral + Kd*derivative;
    //filtro de saturacion
    if(output > MAX)  output = MAX;
    else if(output < MIN)  output = MIN;
     pre_error = error; //actualiza error
     return output;
}
```

Simulación del control del nivel de un depósito con PID

- Se desea controlar el nivel de un depósito actuando sobre el caudal de entrada



$$q_{in}(t) - q_{out}(t) = A\frac{dh(t)}{dt}$$

$$q_{out}(t) = a\sqrt{2gh(t)}$$

$$q_{in}(t) = k \cdot u(t)$$

$$\frac{dh(t)}{dt} = -\frac{a}{A}\sqrt{2gh(t)} + \frac{k}{A}u(t)$$

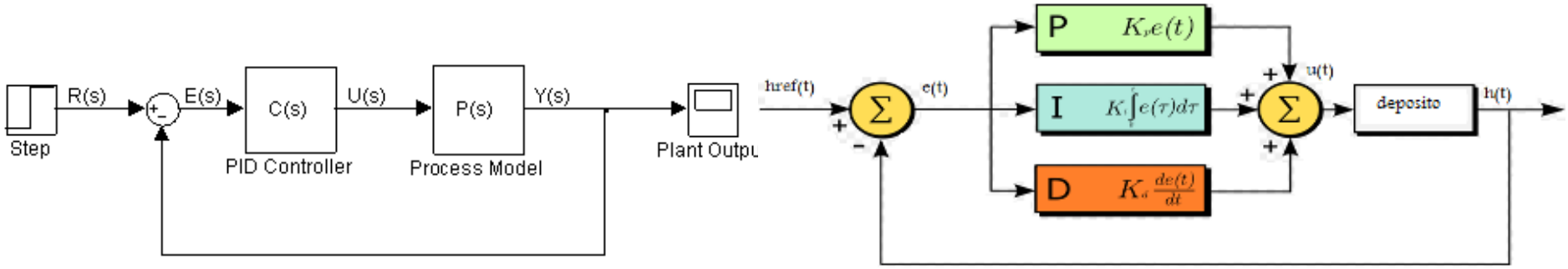
$$e(t) = h_{ref}(t) - h(t)$$

Simulación del control del nivel de un depósito con PID

- Se resuelve la ecuación diferencial utilizando cualquier método visto antes (euler) y se aplica al resultado el control PID

$$\frac{dh(t)}{dt} = -\frac{a}{A}\sqrt{2gh(t)} + \frac{k}{A}u(t)$$

$$e(t) = h_{ref}(t) - h(t)$$



Código de control del nivel de un depósito con PID

```
/* g: Gravedad 9.8 m/s2.
   R: Radio de la circunferencia que define el depósito
   r: Radio de la circunferencia que define las tuberías
   A: Área del depósito. A= pi R^2 (m2).
   a: Área de las tuberías.a= pi r ^2 (m2).
   k: Coeficiente de la valvula    */

#include <stdio.h>          //I/O library
#include <math.h>           //Math library

#define  NDERIVS  1                  //Numero de derivadas
void CalcDerivs(int numDerivs, double *x, double t, double *k1);
float PIDcal(float setpoint,float actual_position, float dt);

#define PI  3.1415   // PI
#define g   9.80665  // constante gravedad terrestre
#define R  10e-2     // radio del deposito
#define r  0.8e-2    // radio de las tuberias de entrada/salida
#define k  1         // coeficiente de valvula
#define HMAX   1     // altura max de tanque
#define Q   0.001    // Caudal entrada m3/seg
double a, A, sc=0;
FILE *output;              /* internal filename */

void CalcDerivs(int numDerivs, double *x, double t, double *k1) {
   if (x[0] < 0) x[0] = 0.0; k1[0]=-a/A*sqrt(2*g*x[0])+u(t);   //Calcula derivada
}
```

Código de control del nivel de un depósito con PID

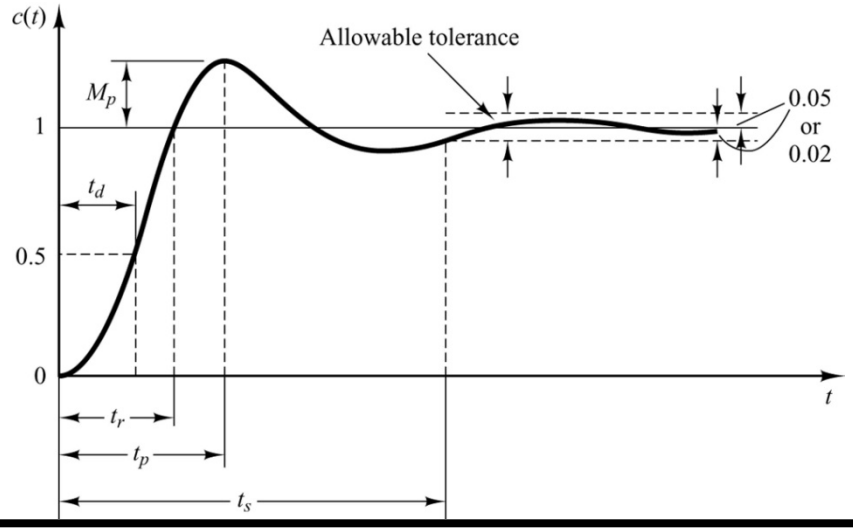
```
double u(double t) {   // funcion para simular entrada de caudal
   return sc*Q/A;
}
int main() {
    double ystar, x[NDERIVS], k1[NDERIVS];
    double h=0.01, t=0.0, tmax=300.0; //Step size, time, final time.
    double sp=0.75;
    int i;                              //contador

   output=fopen("outpid.txt", "w");                        /* fichero externo*/
    x[0]=0.0;                 //nivel inicial
    ystar=x[0];
     a = PI*r*r; A = PI*R*R;
    for (t=0; t<=tmax; t+=h) {
          CalcDerivs(NDERIVS, x, t, k1);
          ystar=x[0];
          if (ystar<0.0) break;
          fprintf(output,"%4.2f \t %g \n",t,ystar);
          for(i=0; i<NDERIVS; i++)
              x[i] = x[i] + h*k1[i];
          if (x[0] >= HMAX) x[0] = HMAX; if (x[0] < 0.0) x[0] = 0.0;
          sc = PIDcal(sp,x[0], h);
    }
    printf("Fin del programa\n"); fclose(output);
    return 0;
}
```

Efecto de las ganancias en controles PID

- Las ganancias de los controles PID tienen los siguientes efectos :
  - Kp reduce el tiempo de subida y el error de estado estacionario sin eliminarlo
  - Ki empeora la respuesta transitoria pero elimina el error de estado estacionario
  - Kd incrementa la estabilidad del sistema reduciendo el sobreimpulso

Efecto de las ganancias en controles PID

Ganancia	Tiempo de subida	Sobreimpulso	Tiempo de establecimiento	Error estado estacionario
Kp ↑	Decrece	Crece	Cambio pequeño	Decrece
Ki	Decrece	Crece	Crece	Elimina
Kd	Cambio pequeño	Decrece	Decrece	Cambio pequeño



Simulación en C

Ajuste de controladores PID

- Para el ajuste de las ganancias de los controles PID se emplean los siguientes métodos:
  - Basados en el error integral

$$\text{Integral of absolute error} \qquad IAE = \int_0^\infty |e(t)|\,dt$$

$$\text{Integral of squared error} \qquad ISE = \int_0^\infty e^2(t)\,dt$$

$$\text{Integral of time weighted absolute error} \qquad ITAE = \int_0^\infty t\,|e(t)|\,dt$$

Ajuste de controladores PID

– Basados en técnicas heurísticas. Ejemplo: método de oscilación Ziegler-Nichols aplicable a plantas estables en lazo abierto:

- Aumentar el valor de K_p hasta que el lazo comience a oscilar (oscilación lineal detectada en la salida del controlador (*u(t)*))
- Registrar la ganancia crítica $K_p = K_c$ y el período de oscilación P_c de *u(t)*, a la salida del controlador
- Ajustar los parámetros del control según la siguiente tabla:

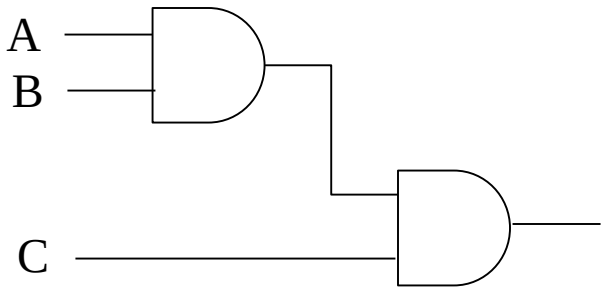
	$\mathbf{K_p}$	$\mathbf{T_r}$	$\mathbf{T_d}$
P	$0.50K_c$		
PI	$0.45K_c$	$\frac{P_c}{1.2}$	
PID	$0.60K_c$	$0.5P_c$	$\frac{P_c}{8}$

Simulación de circuitos lógicos combinacionales

- Es fácil obtener la tabla de verdad de circuitos lógicos combinacionales sencillos
- Se utiliza para ello la representación más sencilla de un entero (short) para no consumir memoria
- Se aplican convenientemente los operadores lógicos
- Se crean tantos ciclos for como señales de entrada existan

Ejemplo de simulación de circuitos lógicos combinacionales

```
/*************************************************************************\
* Programa: tvsd.c                                                        *
* Descripción: Programa que imprime la tabla de verdad de expresiones     *
*              lógicas (circuitos logicos combinacionales)                *
* Expresiones: X = ABC                                                    *
*              X = ABC + A!B + B!C)                                       *
* Autor: Pedro Corcuera                                                   *
* Revisión: 1.0 4/02/2008                                                 *
\*************************************************************************/
#include <stdio.h>

int main(void)
{
  short X, A, B, C;

  printf("Tabla de verdad para X = ABC\n\n");
  printf(" ABC   X\n--------\n");

  for (A=0; A<=1 ;A++) {
     for (B=0; B<=1 ;B++) {
         for (C=0; C<=1 ;C++) {
             X = A && B && C;
             printf(" %d%d%d   %d\n",A, B, C, X);
         }
     }
  }
```



Ejemplo de simulación de circuitos lógicos combinacionales

```
  printf("\n                                        _    _");
  printf("\nTabla de verdad para X = ABC + AB + BC\n\n");
  printf(" ABC   X\n--------\n");

  for (A=0; A<=1 ;A++) {
     for (B=0; B<=1 ;B++) {
         for (C=0; C<=1 ;C++) {
             X = (A && B && C) || (A && !B) || (B && !C);
             printf(" %d%d%d   %d\n",A, B, C, X);
         }
     }
  }
  return 0;
}
```

Estructuras de datos en C
Listas y Colas en C

Índice

- Estructuras autoreferenciadas
- Asignación dinámica de memoria.
- Listas enlazadas.
- Pilas.
- Colas.
- Arboles

Punteros a estructuras

- Son muy frecuentes en C. Se declaran de la misma forma que un puntero a una variable ordinaria. Ej.:

  struct alumno *palumno;

  ALUMNO *palumno; /* usando typedef */
- Para acceder a un campo de la estructura:

  prom_nota = (*palumno).nota

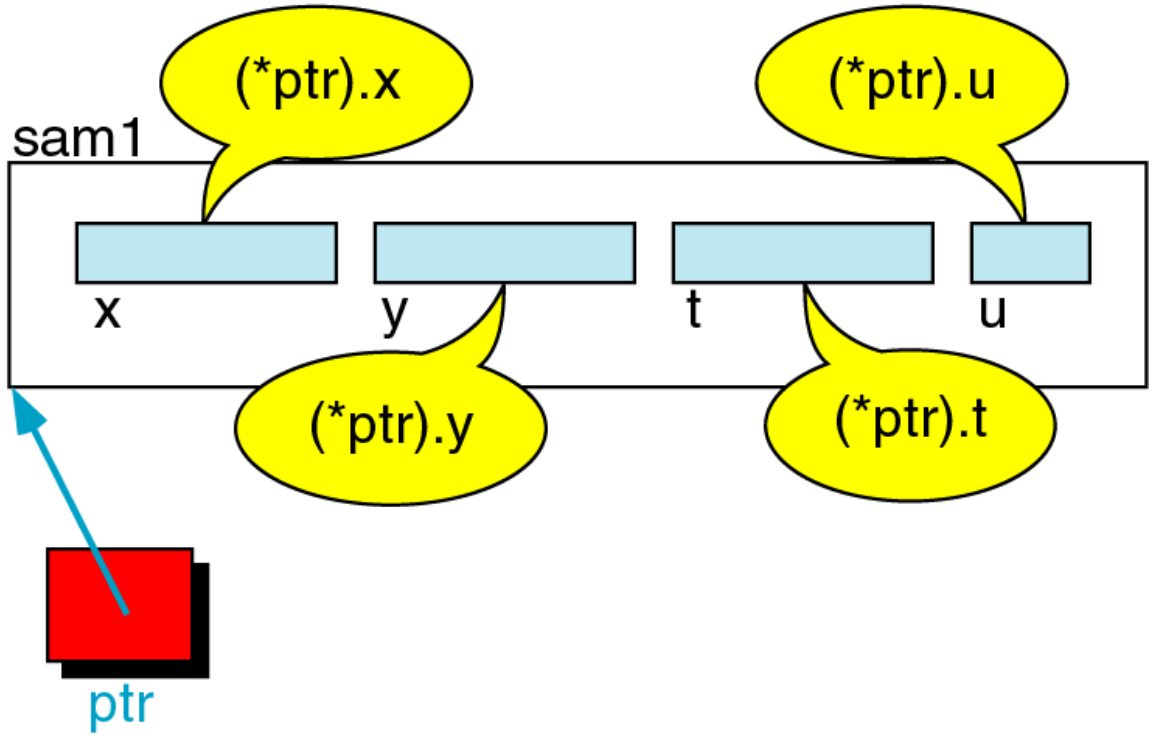
  los paréntesis son estrictamente necesarios.

  Hay un operador (->) alternativo para este propósito:

  palumno -> nota

Punteros a estructuras

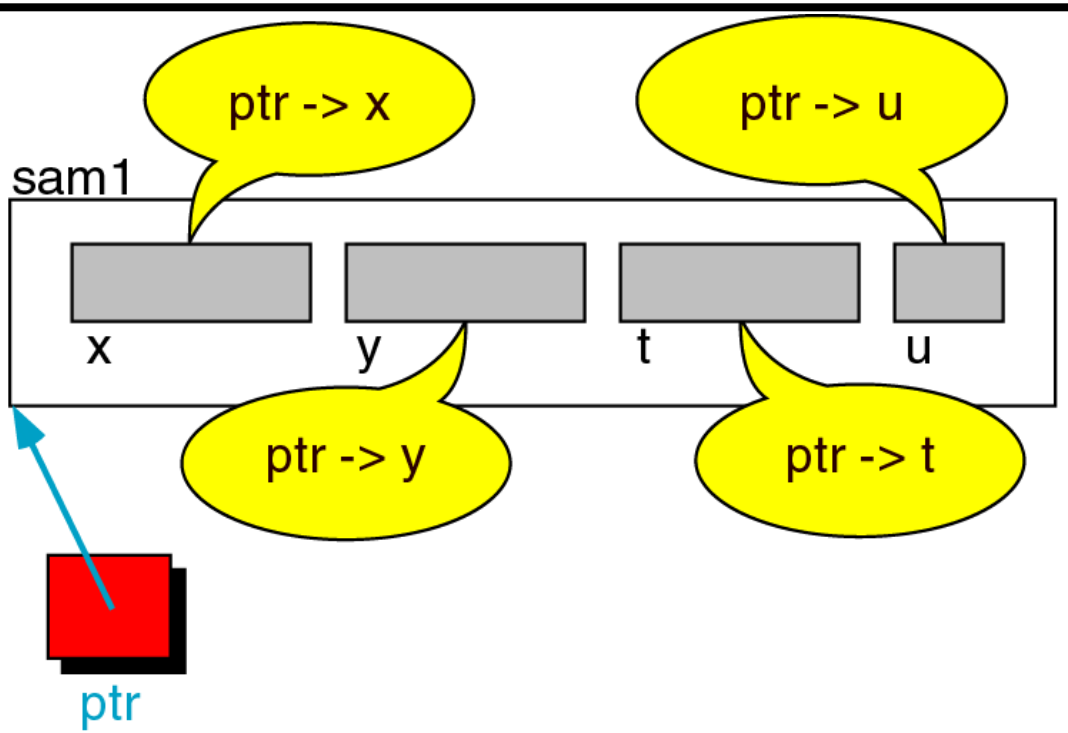
```
typedef struct
  {
   int   x;
   int   y;
   float t;
   char  u;
  } SAMPLE;
   ...
   SAMPLE  sam1;
   SAMPLE  *ptr;
   ...
   ptr = &sam1;
   ...
```

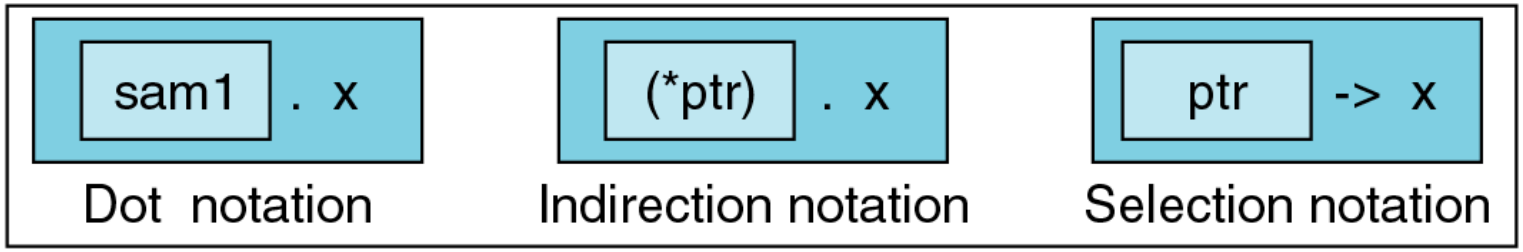


Two ways to reference x

Punteros a estructuras

```
typedef struct
  {
   int   x;
   int   y;
   float t;
   char  u;
  } SAMPLE;
  ...
  SAMPLE   sam1;
  SAMPLE  *ptr;
  ...
  ptr = &sam1;
  ...
```



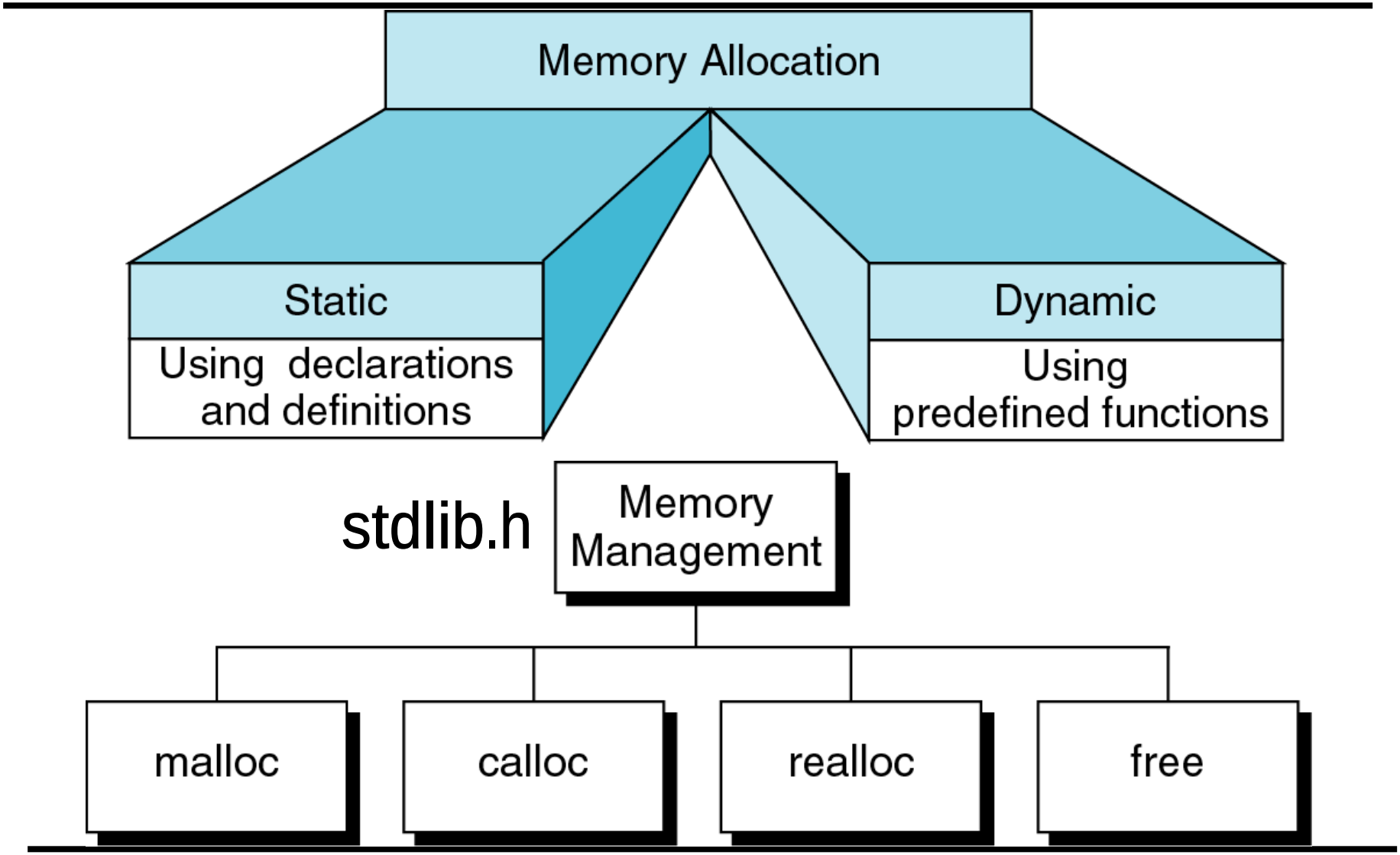


three ways to reference the field **x**

Asignación dinámica de memoria

- Además de la reserva de espacio estática (cuando se declara una variable), es posible reservar memoria de forma dinámica.
- Hay funciones de gestión de memoria dinámica (stdlib.h):
  - void *malloc(size_t): Reserva memoria dinámica.
  - void *calloc(size_t): Reserva memoria dinámica.
  - void *realloc(void *,size_t): Ajusta el espacio de memoria dinámica.
  - free(void *): Libera memoria dinámica.

Memoria dinámica

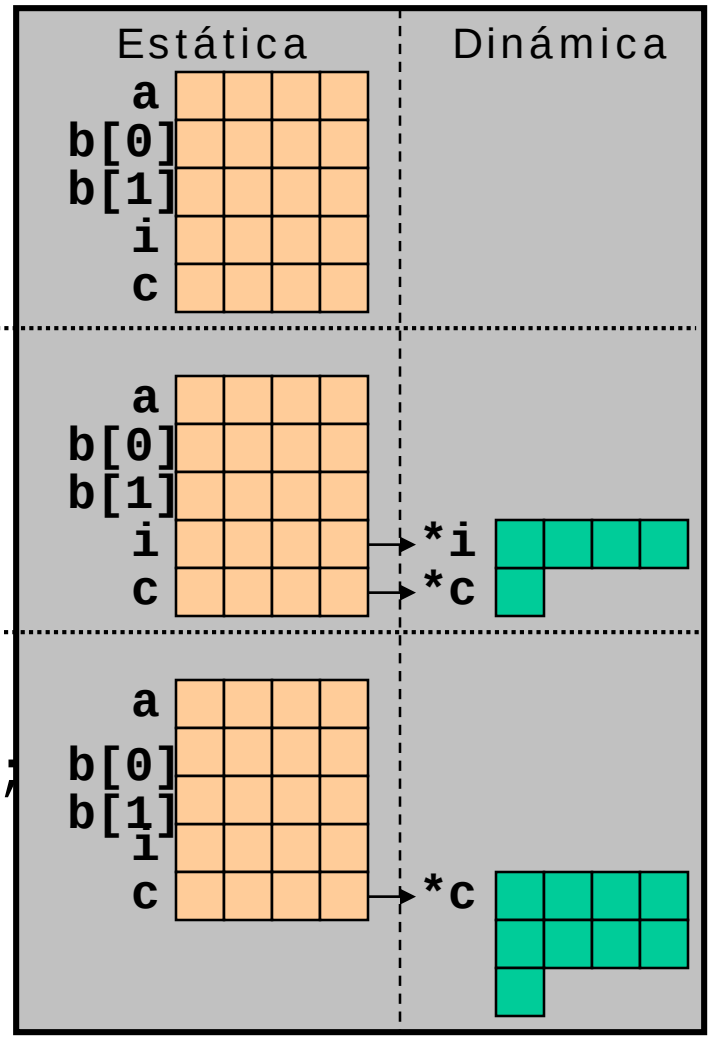


Memoria Dinámica - uso

```
#include <stdlib.h>
int    a,b[2];
int *i;
char *c;
```

```
i=(int *)malloc(sizeof(int));
c=(char *)malloc(sizeof(char));
```

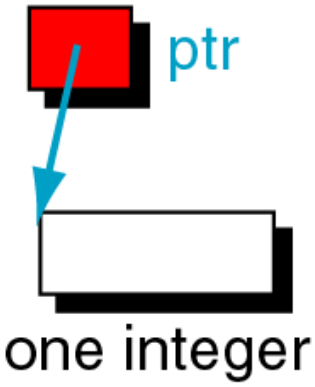
```
free(i);
c=(char *)realloc(c,sizeof(char)*9);
```



Memoria Dinámica – uso malloc

`void *malloc(size_t tamaño_bloque)`

devuelve NULL si no hay suficiente memoria

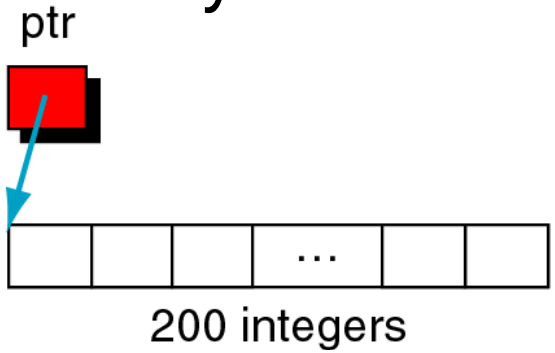



```
ptr = (int *)malloc(sizeof(int));
if ( ptr == NULL)
  /* No hay memoria */
  printf ("ERROR. No hay memoria\n");
else
  …. /* Memoria disponible */
```

Memoria Dinámica – uso calloc

`void *calloc(size_t nelem, size_t elsize)`

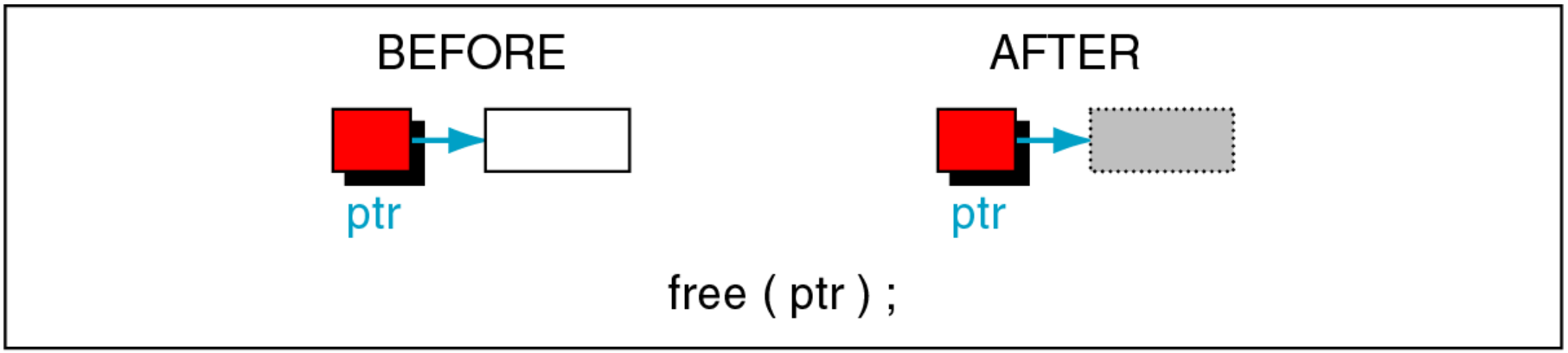
devuelve NULL si no hay suficiente memoria

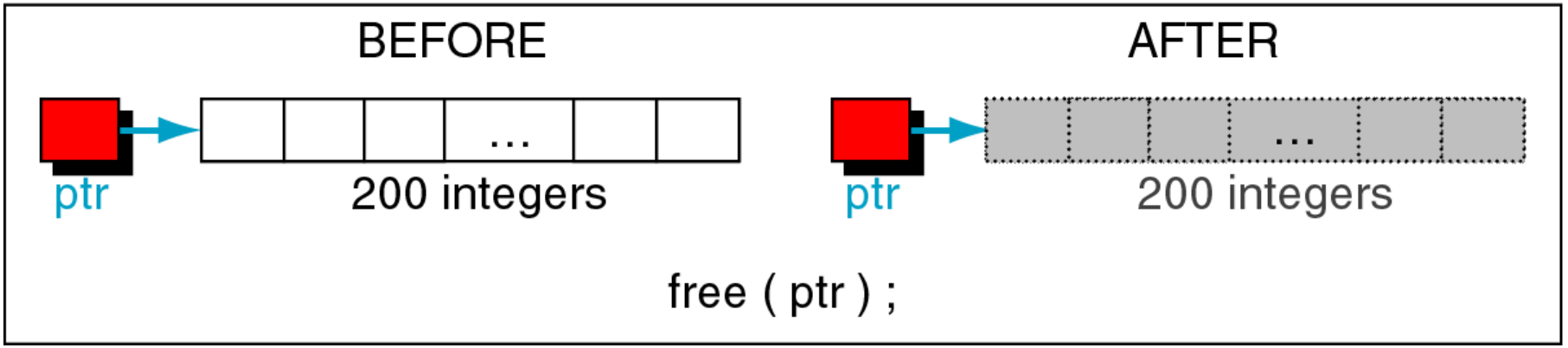



```
ptr = (int *)calloc(200, sizeof(int));
if ( ptr == NULL)
  /* No hay memoria */
  printf ("ERROR. No hay memoria\n");
else
  …. /* Memoria disponible */
```

Memoria Dinámica – uso free

```
void free(void *puntero_al_bloque)
```





Ejemplo: array dinámico

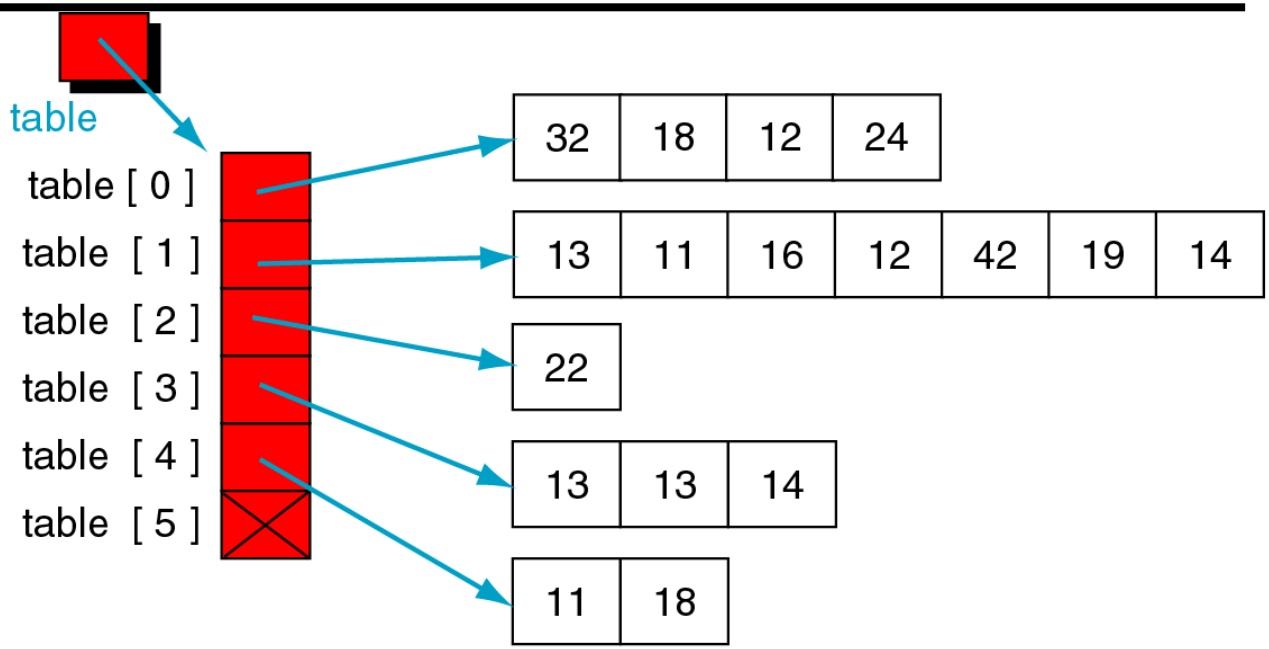
```
/*****************************************************************\
* Programa: array_dinamico.c                                      *
* Descripción: Prog. que crea una array dinamico y genera otro con los   *
*              elementos pares del primer array y lo imprime      *
\*****************************************************************/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int dim_usu;   /* dimension del vector del usuario */
   int dim_par;   /* dimensión del vector de elementos pares */
   int n;         /* indice para los for */
   int m;         /* indice para recorrer arrya de pares */
   int *pvec_usu; /* puntero al vector introducido por el usuario */
   int *pvec_par; /* puntero al vector elementos pares (dinamico) */

   printf("Introduzca la dimension del vector: ");
   scanf(" %d", &dim_usu);

   pvec_usu = (int *) calloc( dim_usu, sizeof(int)); /*Asignar memoria vect. usuario
   */
   /*pvec_usu = (int *) malloc( dim_usu*sizeof(int));*/
   if (pvec_usu == NULL) {  /* si no hay memoria */
         printf("Error: no hay memoria para un vector de %d elementos\n", dim_usu);
   }
```

Ejemplo: array dinámico

```
    else
    {
          for (n = 0; n < dim_usu; n++) /* pedir elementos del vector */
          { printf("Elemento %d = ", n); scanf("%d", &(pvec_usu[n]));}
          dim_par = 0;
          for (n = 0; n < dim_usu; n++)
             if ((pvec_usu[n] % 2) == 0) dim_par++;
          pvec_par = (int *) calloc( dim_par, sizeof(int));
          if (pvec_par == NULL) {  /* si no hay memoria */
             printf("Error: no hay memoria para un vector de %d elementos\n",
    dim_par);
          }
          else
          { /* se copian los elementos pares */
                    m = 0;
                    for (n = 0; n < dim_usu ; n++)
                        if ((pvec_usu[n] % 2) == 0) { pvec_par[m] = pvec_usu[n]; m++;}
                    printf("\n----------------------------\n");
                    for (n = 0; n < dim_par ; n++)
                              printf("Elemento par %d = %d \n", n, pvec_par[n]);
          }
          free (pvec_par);
    }
    free (pvec_usu);
}
```

Matriz con número de columnas diferentes

- Se representa como un array de arrays (puntero a puntero) dinámico.

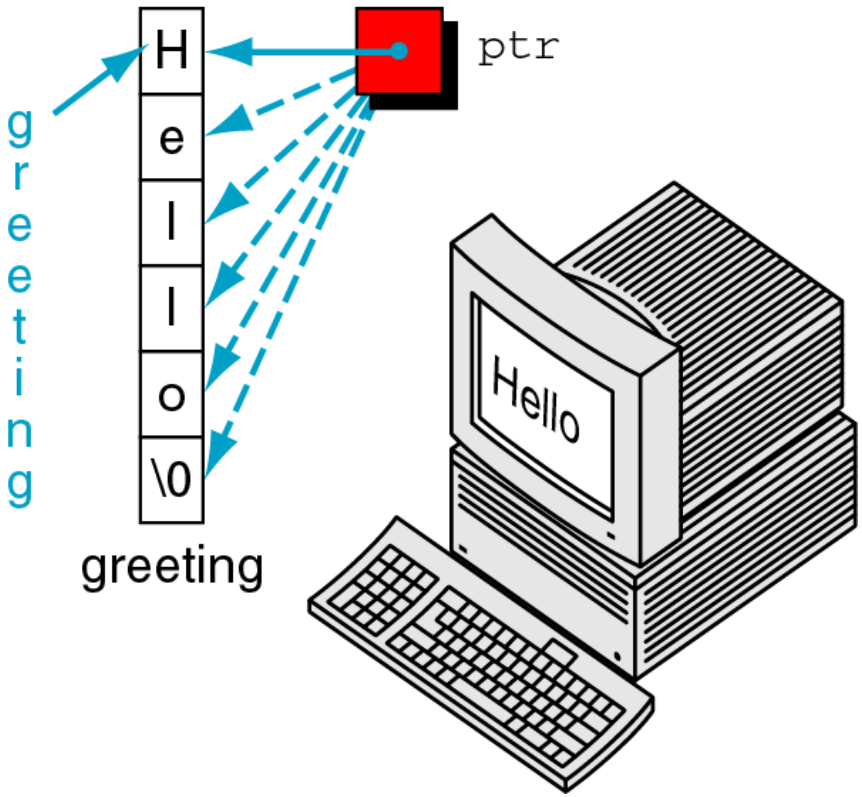



```
table = (int **)calloc (rowNum + 1, sizeof(int*));

table[0] = (int*)calloc (4, sizeof(int));
table[1] = (int*)calloc (7, sizeof(int));
table[2] = (int*)calloc (1, sizeof(int));
table[3] = (int*)calloc (3, sizeof(int));
table[4] = (int*)calloc (2, sizeof(int));
table[5] = NULL;
```

Punteros a cadenas de caracteres

- Utiles en recorridos de cadenas de caracteres.

```
{ /* Printing Strings */
  char  greeting[] = "Hello";
  char *ptr;

  ptr = greeting;
  while (*ptr != '\0')
       {

      printf( "%c", *ptr);
      ptr++;
     } /* while */
  print("\n");
} /* Printing Strings */
```



Estructuras autoreferenciadas

- Para construir estructuras de datos más potentes es necesario incluir *dentro* de una estructura un campo que sea un puntero a la misma estructura.

```
struct alumno
{
   char dni[10];
   char nombre[100];
   struct fecha fnac;
   float notas[10];
   struct alumno *palumno;
};
```

Estructuras autoreferenciadas

- Estructuras que contienen uno o más punteros a una estructura del mismo tipo.
- Pueden ser enlazadas entre ellas para formar estructuras de datos útiles tales como listas, colas, pilas y árboles.
- Terminan con un puntero NULL.

```
struct nodo
{
  int dato;   /* campos necesarios */
  struct nodo *sig; /* puntero de enlace*/
};
struct nodo *pnodo;
```

Estructuras autoreferenciadas

```
struct Info {
    int item1;
    char *item2;
    float item3;
};
struct Nodo {
    struct Info info;
    struct Nodo *sig;
};
typedef struct Nodo *TipoLista;
TipoLista lista_vacia(void) ;
```

Asignación dinámica de memoria

- El propósito es obtener y liberar memoria durante la ejecución.
- Con la función malloc se reserva el número de bytes de memoria requerida. Devuelve un puntero (void *)
- Para determinar el tamaño de un nodo se usa el operador sizeof.
- Ejemplo:

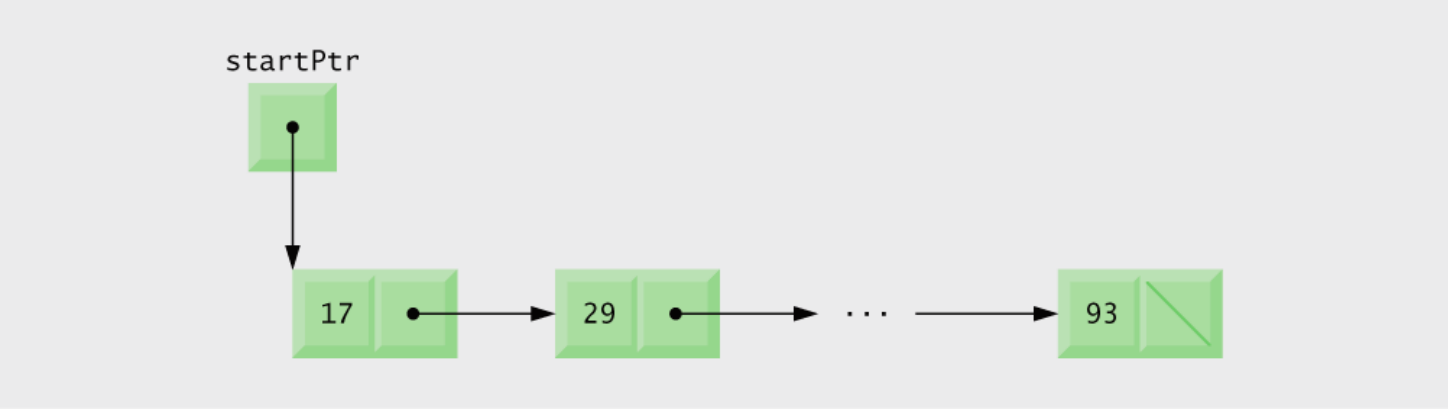
```
struct nodo *p;
p=(struct nodo *)malloc(sizeof(struct nodo));
```

Listas enlazadas

- Colección lineal de nodos autoreferenciados.
- Conectados por punteros enlazados.
- Acceso mediante un puntero al nodo inicial de la lista.
- Los nodos siguientes se acceden mediante el campo de puntero de enlace del nodo en curso.
- El puntero de enlace del último nodo se asigna a NULL para indicar final de la lista.
- Las listas enlazadas se usan en lugar de un array cuando se tiene un número impredecible de elementos de datos y cuando se requiere ordenar rápidamente la lista.

Listas enlazadas

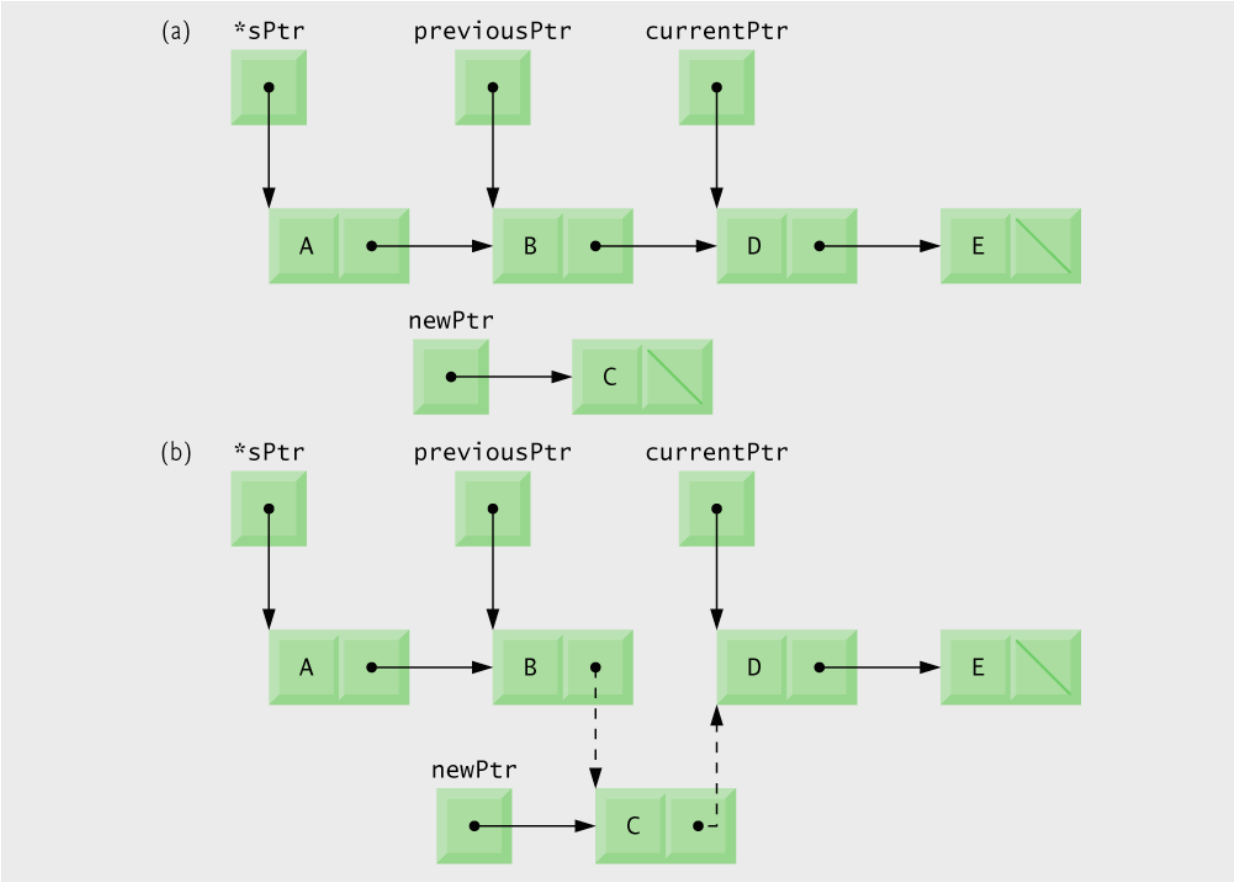
- Representación gráfica de una lista enlazada de números.



- Las operaciones sobre listas ordenadas más comunes son:
  - Crear_lista: inicializa lista a vacío.
  - Insertar: añade un elemento (nodo) a la lista.
  - Eliminar: suprime el nodo que contiene un elemento especificado de la lista.
  - ImprimeLista: imprime todos los elementos de la lista.
  - ListaVacía: operación booleana que indica si la lista está vacía.

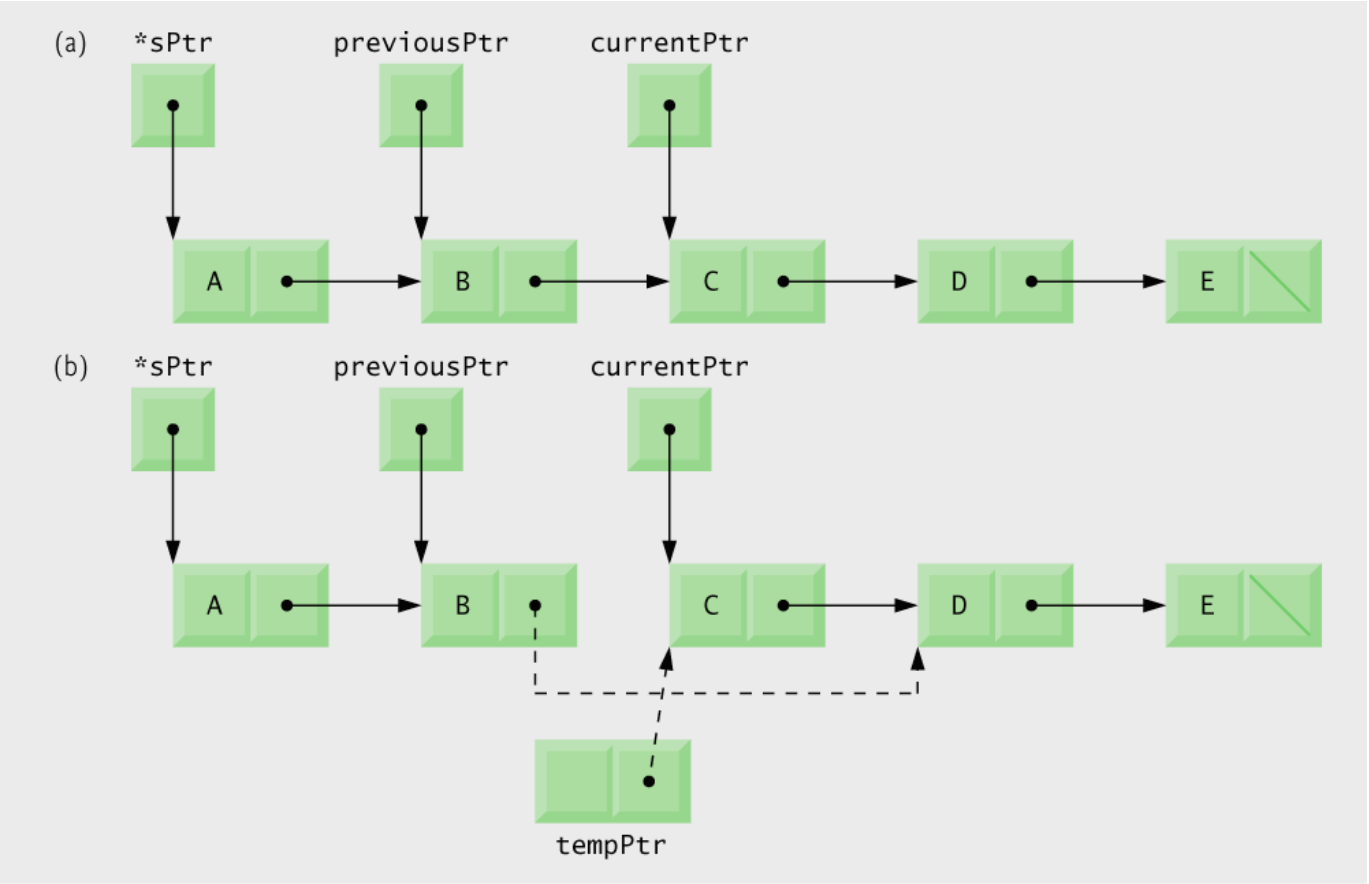
Listas enlazadas – Inserción de un nodo

- Representación gráfica de la inserción de un nodo en una lista ordenada.



Listas enlazadas – Eliminación de un nodo

- Representación gráfica de la eliminación de un nodo en una lista ordenada.



Listas enlazadas – Declaración

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

struct Info {
    int item1;
    char *item2;
    float item3;
};

struct Nodo {
    struct Info info;
    struct Nodo *sig;
};

typedef struct Nodo *TipoLista;
```

Listas enlazadas – Declaración

```
TipoLista lista_vacia(void) ;
void lee_nodo(struct Info *inf);
void lee_nodof(struct Info *inf, FILE *fp);
void muestra (TipoLista lista);
TipoLista inserta (TipoLista lista);
void consulta_item2 (TipoLista lista);
void modifica_item2 (TipoLista lista);
TipoLista borra_item2 (TipoLista lista);
void guarda (TipoLista lista);
TipoLista recupera(TipoLista lista);
```

Listas enlazadas – Funciones

```
TipoLista lista_vacia (void) {
    return NULL;
}

void lee_nodo(struct Info *inf) {
    int i;
    char w[20], *pw;
    float x;

    printf("Ingresa entero: ");
    scanf(" %d", &i);
    printf("Ingresa palabra: ");
//    scanf(" %s", w);  // cadena simple
    scanf(" %[^\n]", w);   // cadena compuesta
    printf("Ingresa flotante: ");
    scanf(" %f", &x);
    inf->item1 = i;
    pw = malloc(strlen(w)+1);
    strcpy(pw,w);
    inf->item2= pw;
    inf->item3 = x;
}
```

Listas enlazadas – Funciones

```
TipoLista inserta (TipoLista lista)
{
    struct Info *inf = malloc (sizeof (struct Info));
    lee_nodo(inf);
    struct Nodo *nuevo = malloc (sizeof (struct Nodo));
    nuevo-> info = *inf;
    nuevo->sig= lista;
    lista = nuevo;
    return lista;
}

void muestra(TipoLista lista)
{
    struct Nodo *aux;
    printf("->");
    for (aux =lista; aux!=NULL; aux =aux->sig) {
        printf ("[%s]-> ",aux->info.item2);
//printf ("[%d, %s, %f]-> \n",aux->info.item1, aux->info.item2, aux->info.item3);
    }
    printf("\n");
}
```

Listas enlazadas – Funciones

```
TipoLista borra_item2 (TipoLista lista)
{
    char w[20];
    struct Nodo *aux, *atras;

    printf("item 2 a borrar: ");
    scanf(" %[^\n]", w);   // cadena compuesta

    for (atras=NULL, aux=lista; aux!=NULL; atras=aux, aux=aux->sig)
        if (strcmp(aux->info.item2, w)==0){
           if (atras==NULL)
                lista = aux-> sig;
            else
                atras->sig= aux-> sig;
            free (aux);
            printf("nodo eliminado\n");
            return lista;
        }
    return lista;
}
```

Listas enlazadas – Funciones

```
void consulta_item2 (TipoLista lista) {
    char w[20];
    struct Nodo *aux;
    printf("item 2 a consultar: ");
    scanf(" %[^\n]", w);   // cadena compuesta
    for (aux=lista; aux!=NULL; aux=aux->sig)
        if (strcmp(aux->info.item2, w)==0){
            printf("item 1: %d\n",aux->info.item1);
            printf("item 3: %.2f\n",aux->info.item3);
        }
}
void modifica_item2 (TipoLista lista){
    char w[20];
    struct Nodo *aux;
    printf("item 2 a modificar: ");
    scanf(" %[^\n]", w);   // cadena compuesta
    for (aux=lista; aux!=NULL; aux=aux->sig)
        if (strcmp(aux->info.item2, w)==0){
            lee_nodo(&aux->info);
        }
}
```

Listas enlazadas – Funciones

```
void guarda(TipoLista lista){
    struct Nodo *aux;
    FILE *fp;
    char c;
    printf("Guardar en fichero binario (b)?: ");
    scanf(" %c", &c);
    if (c=='b')
        fp = fopen("itemsb.dat", "wb");     // fichero binario
    else
        fp = fopen("itemst.dat", "w");     // fihcero texto
    if(fp == NULL)
    {
      printf("Error en escritura de fichero");
      exit(-1);
    }
    for (aux=lista; aux!=NULL; aux=aux->sig)
    {
        if (c=='b'){
             printf("%d  \n",aux->info.item1);
            fwrite(&aux->info, sizeof(struct Info), 1,fp);  // corregir error de grabacion de
    campos char *
        }
        else{
            fprintf(fp,"%d\n",aux->info.item1);
            fprintf(fp,"%s\n",aux->info.item2);
            fprintf(fp,"%.2f\n",aux->info.item3);
        }
    }
    fclose(fp);
    printf("fichero creado\n");
}
```

Listas enlazadas – Funciones

```
void lee_nodof(struct Info *inf, FILE *fp)
{
    int i;
    char w[20], *pw;
    float x;

    fscanf(fp," %d", &i);

    fscanf(fp," %[^\n]", w);   // cadena compuesta
    fscanf(fp," %f", &x);
    inf->item1 = i;
    pw = malloc(strlen(w)+1);
    strcpy(pw,w);
    inf->item2= pw;
    inf->item3 = x;
}
```

Listas enlazadas – Funciones

```
TipoLista recupera(TipoLista lista){
    FILE *fp;
//    struct Nodo *n;
    char c;
    printf("Fichero binario (b)?: ");
    scanf(" %c", &c);
    if (c=='b')
        fp = fopen("itemsb.dat", "rb");
    else
        fp = fopen("itemst.dat", "r");
    if(fp == NULL)
    {
      printf("Error en lectura de fichero");
      exit(-1);
    }
    while (!feof(fp))
    {
        struct Info *inf = malloc (sizeof (struct Info));
        if (c=='b')
            fread(inf, sizeof(struct Info), 1, fp);
        else
            lee_nodof(inf, fp);
        struct Nodo *nuevo = malloc (sizeof (struct Nodo));
        nuevo-> info = *inf;
        nuevo->sig= lista;
        lista = nuevo;
    }
    fclose(fp);
    printf("lista creada\n");
    return lista->sig;
}
```

Listas enlazadas – main

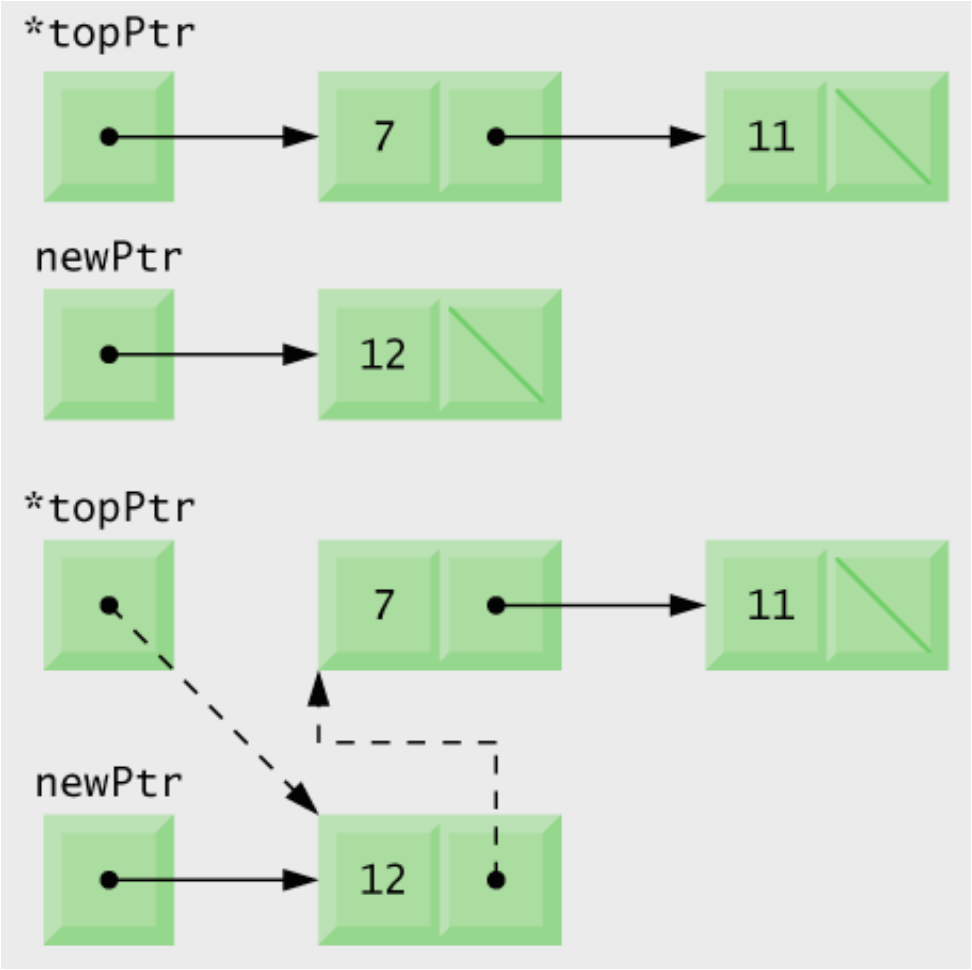
```
int main( )
{
    TipoLista l; int opcion=0, salir=0; char continuar[2];
    l =lista_vacia();   // inicializacion lista
    do {
        printf("\n MENU DE LA APLICACION: \n");
        printf("\t1) Mostrar\n"); printf("\t2) Insertar\n");
        printf("\t3) Consultar\n"); printf("\t4) Modificar\n");
        printf("\t5) Borrar\n"); printf("\t6) Guardar\n");
        printf("\t7) Recuperar\n"); printf("\t8) Salir\n");
        printf("  Elige una opcion del menu: "); scanf(" %d",&opcion);
        switch(opcion){
            case 1: muestra(l); break;         //mostrar
            case 2: //insertar
                do {
                    l = inserta(l); printf("Desea continuar (si/no): "); scanf(" %s", continuar);
                } while (strcmp(continuar, "no")!=0); break;
            case 3: consulta_item2(l); break; //consultar
            case 4: modifica_item2(l); break; //modificar
            case 5: borra_item2(l);break;     //Borrar
            case 6: guarda(l); break; //guardar
            case 7: l=recupera(l); break; //recuperar
            case 8: salir = 1; break;
            default:
                printf("\nOpcion incorrecta\n");
        } // switch
    } while (salir != 1);
    printf("\nPrograma terminado\n");
    return 0;
}
```

Pilas

- Son listas tipo LIFO (Last Input First Output).
- Los nuevos nodos se insertan y se eliminan sólo en la cima.
- El último nodo de la pila se indica por un enlace a NULL.
- Las operaciones sobre pilas más comunes son:
  - push: añadir un nuevo nodo en la cima de la pila.
  - pop: elimina un nodo de la cima.

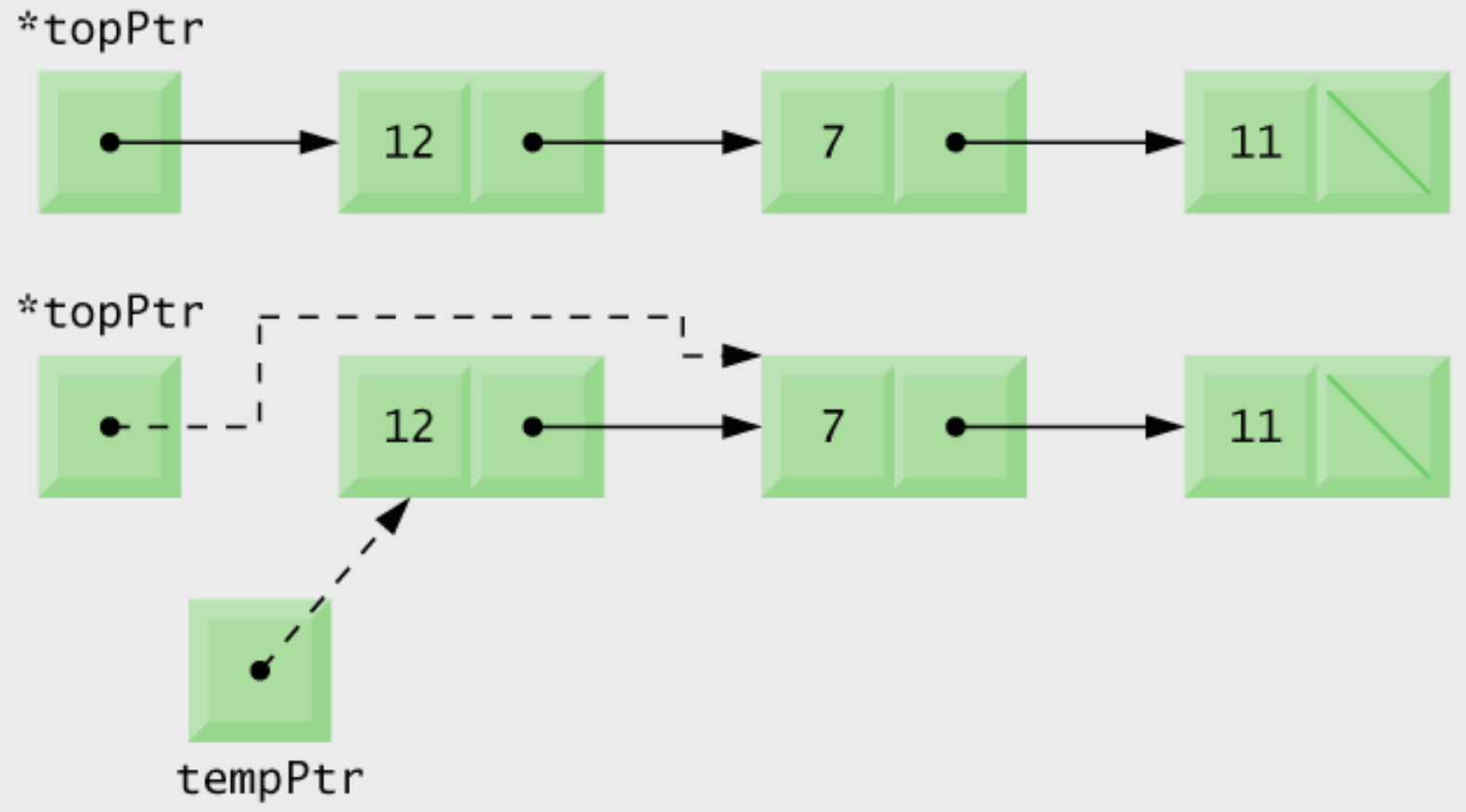
Pilas – Operación push

- Añadir un nuevo nodo en la cima de la pila



Pilas – Operación pop

- Elimina un nodo en la cima de la pila

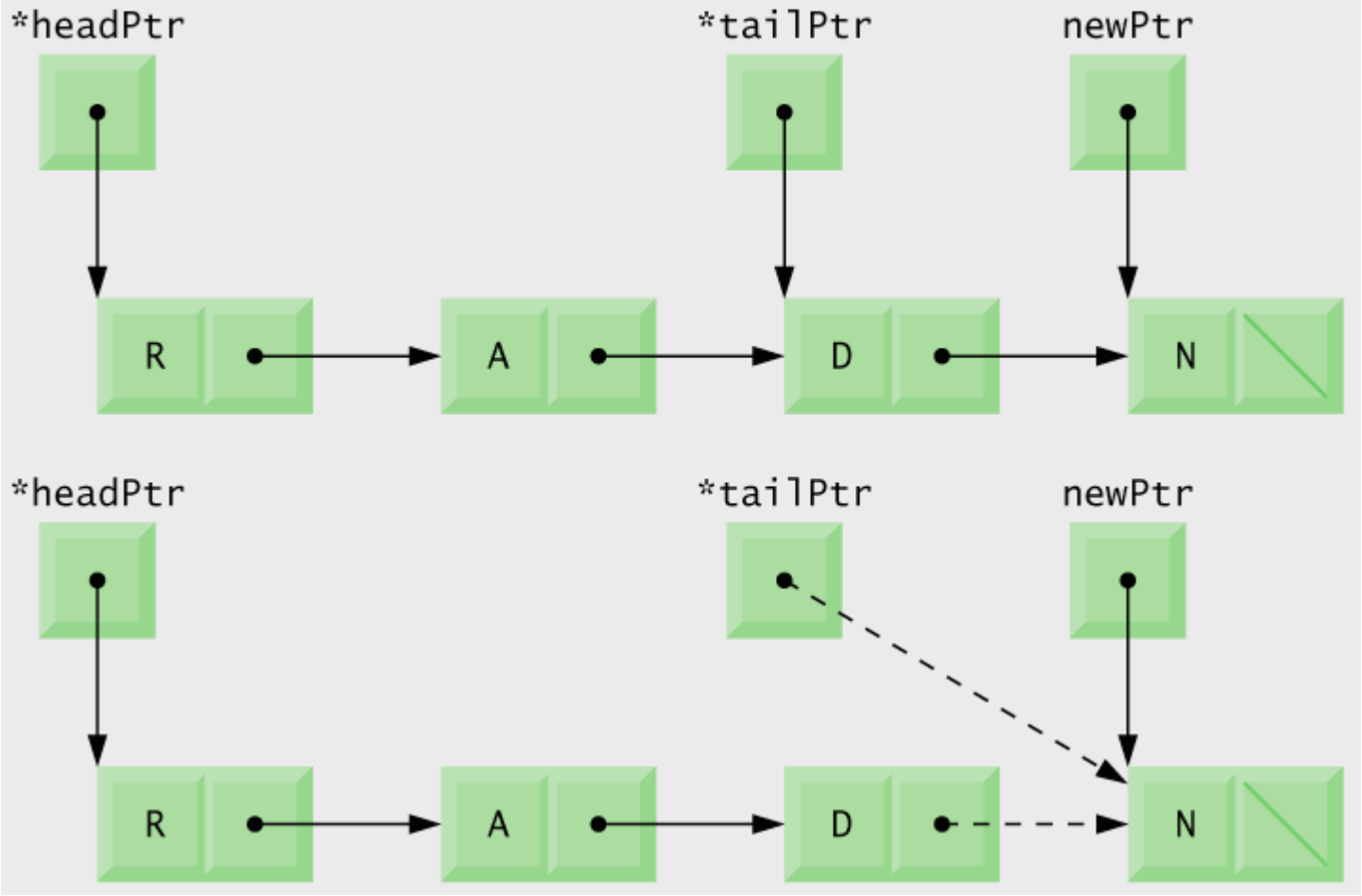


Colas

- Son listas tipo FIFO (First Input First Output).
- Similar a una cola habitual de un banco, servicio, etc.
- Los nuevos nodos se insertan por la cola.
- Los nodos se eliminan por el frente.
- Las operaciones sobre colas más comunes son:
  - encolar: insertar un nuevo nodo por la cola.
  - desencolar: eliminar el nodo del frente.
- Representación gráfica de una cola:

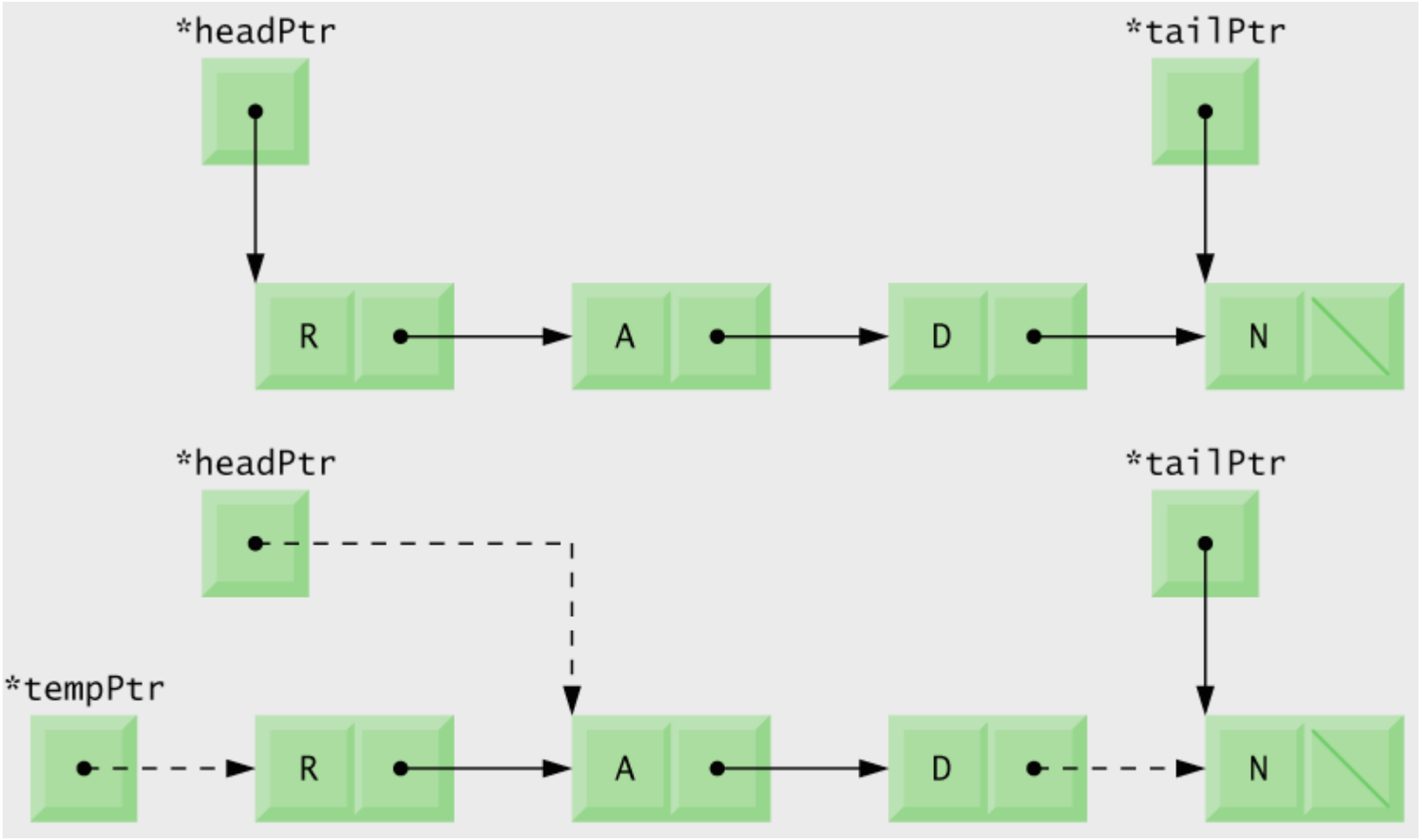
Colas – Operación encolar

- Inserta un nodo por la cola.



Colas – Operación desencolar

- Elimina un nodo por el frente.



Simulación de colas

- Se puede simular una cola real teniendo en cuenta la frecuencia de arribos y el tiempo de simulación. En los nodos se almacenará el tiempo de arribo.
- Para la gestión de la cola se sigue el siguiente algoritmo:
  - Se considera que el tiempo avanza en incrementos de minutos.
  - Cada minuto se comprueba si arriba un nuevo cliente.
  - Si arriba un cliente y la cola no esta llena, se añade el cliente a la cola.
  - Para añadir un cliente a la cola: se crea una estructura Item con el tiempo de arribo del cliente y el tiempo requerido por el cliente.
  - Si la cola está llena se rechaza el cliente.
  - Para elaborar la estadística se registra el número total de clientes y el número de clientes no atendidos.

Simulación de colas

- Procesar el frente de la cola: si la cola no está vacía y si el servidor no está ocupado con un cliente anterior, eliminar el item o nodo del frente de la cola. Como el nodo contiene el tiempo cuando el cliente entró en la cola, por diferencia con el tiempo en curso se obtiene el número de minutos que el cliente ha estado en la cola. El item también contiene el número de minutos requeridos por el cliente, que determina cuánto tiempo el servidor estará ocupado con el nuevo cliente. Usar una variable para seguir la pista del tiempo de espera.
- Si el servidor está ocupado, no se elimina ningún nodo de la cola. La variable que mantiene el tiempo de espera debe ser decrementada.

Simulación de cola

```
#include <stdio.h>
#include <stdlib.h>    // for rand() and srand()
#include <time.h>      // for time()
#include "queue.h"     // change Item typedef
#define MIN_PER_HR 60.0
bool newcustomer(double x);   // is there a new customer?
Item customertime(long when); // set customer parameters
int main(void)
{
    Queue line;
    Item temp;                // new customer data
    int hours;                // hours of simulation
    int perhour;              // average # of arrivals per hour
    long cycle, cyclelimit;   // loop counter, limit
    long turnaways = 0;       // turned away by full queue
    long customers = 0;       // joined the queue
    long served = 0;          // served during the simulation
    long sum_line = 0;        // cumulative line length
    int wait_time = 0;        // time until Sigmund is free
    double min_per_cust;      // average time between arrivals
    long line_wait = 0;       // cumulative time in line

    InitializeQueue(&line);
    srand((unsigned int) time(0)); // random initializing of rand()
    puts("Caso de uso: Cola en gran superficie");
    puts("Ingresa el numero de horas de simulacion:");
    scanf("%d", &hours);
    cyclelimit = MIN_PER_HR * hours;
    puts("Ingresa el numero promedio de clientes por hora:");
    scanf("%d", &perhour);
    min_per_cust = MIN_PER_HR / perhour;
```

Simulación de cola

```
    for (cycle = 0; cycle < cyclelimit; cycle++) {
        if (newcustomer(min_per_cust)) {
            if (QueueIsFull(&line))
                turnaways++;
            else {
                customers++;
                temp = customertime(cycle);
                EnQueue(temp, &line);
            }
        }
        if (wait_time <= 0 && !QueueIsEmpty(&line)) {
            DeQueue (&temp, &line);
            wait_time = temp.processtime;
            line_wait += cycle - temp.arrive;
            served++;
        }
        if (wait_time > 0)
            wait_time--;
        sum_line += QueueItemCount(&line);
    }
    if (customers > 0) {
        printf("clientes aceptados: %ld\n", customers);
        printf(" clientes servidos: %ld\n", served);
        printf("         abandonos: %ld\n", turnaways);
        printf(" tamano medio cola: %.2f\n", (double) sum_line / cyclelimit);
        printf("tiempo espera medio: %.2f minutes\n",(double) line_wait / served);
    }
    else puts("Sin clientes!");
    EmptyTheQueue(&line);
    puts("Fin"); return 0;
}
```

Simulación de cola

```
// x = average time, in minutes, between customers
// return value is true if customer shows up this minute
bool newcustomer(double x)
{
    if (rand() * x / RAND_MAX < 1)
        return true;
    else
        return false;
}

// when is the time at which the customer arrives
// function returns an Item structure with the arrival time
// set to when and the processing time set to a random value
// in the range 1 - 3
Item customertime(long when)
{
    Item cust;

    cust.processtime = rand() % 3 + 1;
    cust.arrive = when;

    return cust;
}
```

Cola - header

```
#ifndef _QUEUE_H_
#define _QUEUE_H_
#include <stdbool.h>

typedef struct item {
    long arrive;      // the time when a customer joins the queue
    int processtime;  // the number of consultation minutes desired
} Item;
#define MAXQUEUE 10
typedef struct node{
    Item item;    struct node * next;
} Node;
typedef struct queue{
    Node * front;  /* pointer to front of queue  */
    Node * rear;   /* pointer to rear of queue   */
    int items;     /* number of items in queue   */
} Queue;

void InitializeQueue(Queue * pq); /* operation:  initialize the queue     */
bool QueueIsFull(const Queue * pq); /* operation:  check if queue is full   */
bool QueueIsEmpty(const Queue *pq); /* operation:  check if queue is empty     */
/* operation:        determine number of items in queue         */
int QueueItemCount(const Queue * pq);
/* operation:        add item to rear of queue                  */
bool EnQueue(Item item, Queue * pq);
/* operation:        remove item from front of queue            */
bool DeQueue(Item *pitem, Queue * pq);
/* operation:        empty the queue                            */
void EmptyTheQueue(Queue * pq);
#endif
```

Cola

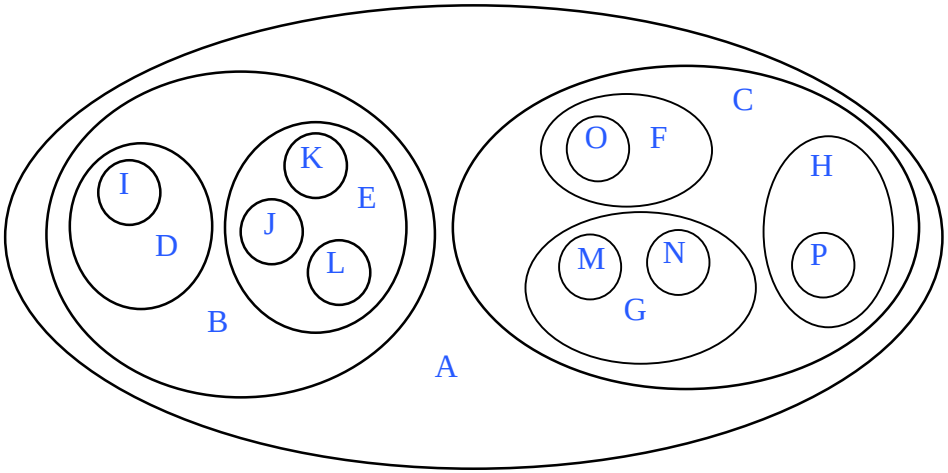
`queue.c`

Arboles

- Son estructuras de datos en las que los nodos contiene dos o más enlaces.
- Definición a nivel lógico: Una estructura de árbol con tipo base T es:
  - La estructura vacía.
  - Un nodo de tipo T con un número finito de estructuras árbol disjuntas asociadas de tipo base T, llamadas subárboles conectadas por ramas o aristas.

Arboles - Representación

- Conjuntos anidados:



- Paréntesis anidados:

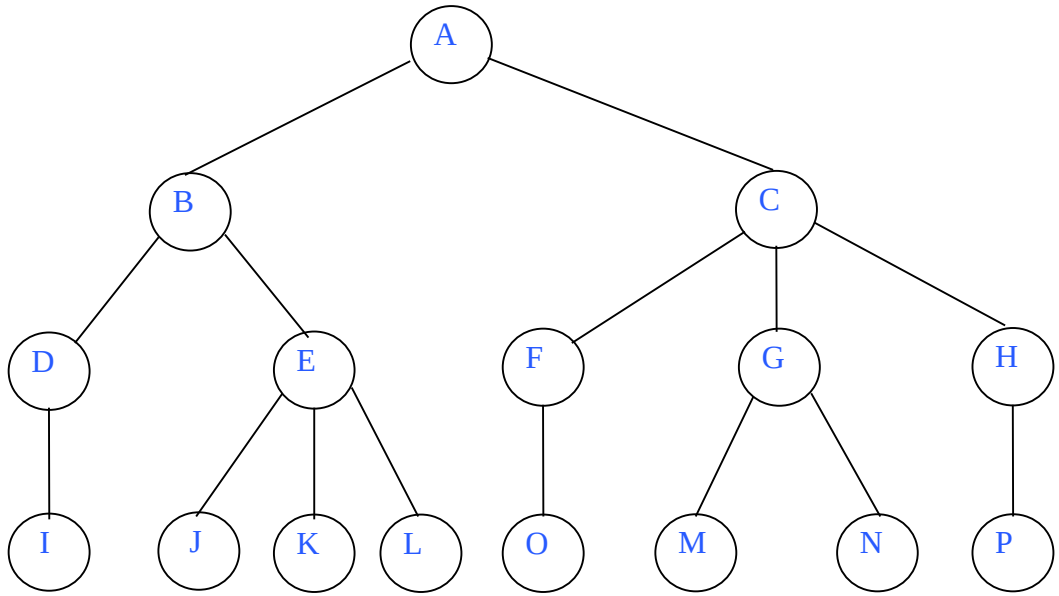
(A (B (D (I), E (J, K, L)), C (F (O), G (M, N), H (P))))

Arboles - Representación

- Indentación:

A
- B
  - D
    - I
  - E
    - J
    - K
    - L
- C
  - F
    - O
  - G
    - M
    - N
  - H
    - P

- Grafo:



La representación más utilizada es la de grafo

Arboles – definiciones de términos

- Tipos de **Nodos**:
  - Nodo Raíz
  - Nodo Descendiente o Hijo
  - Nodo hoja o Interior
  - Nodo antecesor o sucesor
- **Altura**: Número de aristas o ramas desde la raíz hasta el nodo hoja más distante desde éste. Por definición el nodo raíz esté en el nivel 1.
- **Profundidad de un nodo**: número de aristas del camino desde la raíz hasta el nodo.

Arboles – definiciones de términos

- **Grado de un nodo**: número de hijos o descendientes de un nodo interior.
- **Grado de un árbol**: máximo grado de los nodos de un árbol.
- Máximo número de nodos en un árbol de altura h y grado d:

$$N_d(h) = 1 + d + d^2 + \ldots + d^{h-1} = \sum_{i=0}^{h-1} d^i$$

- Para d=2 (árbol binario):

$$N_2(h) = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

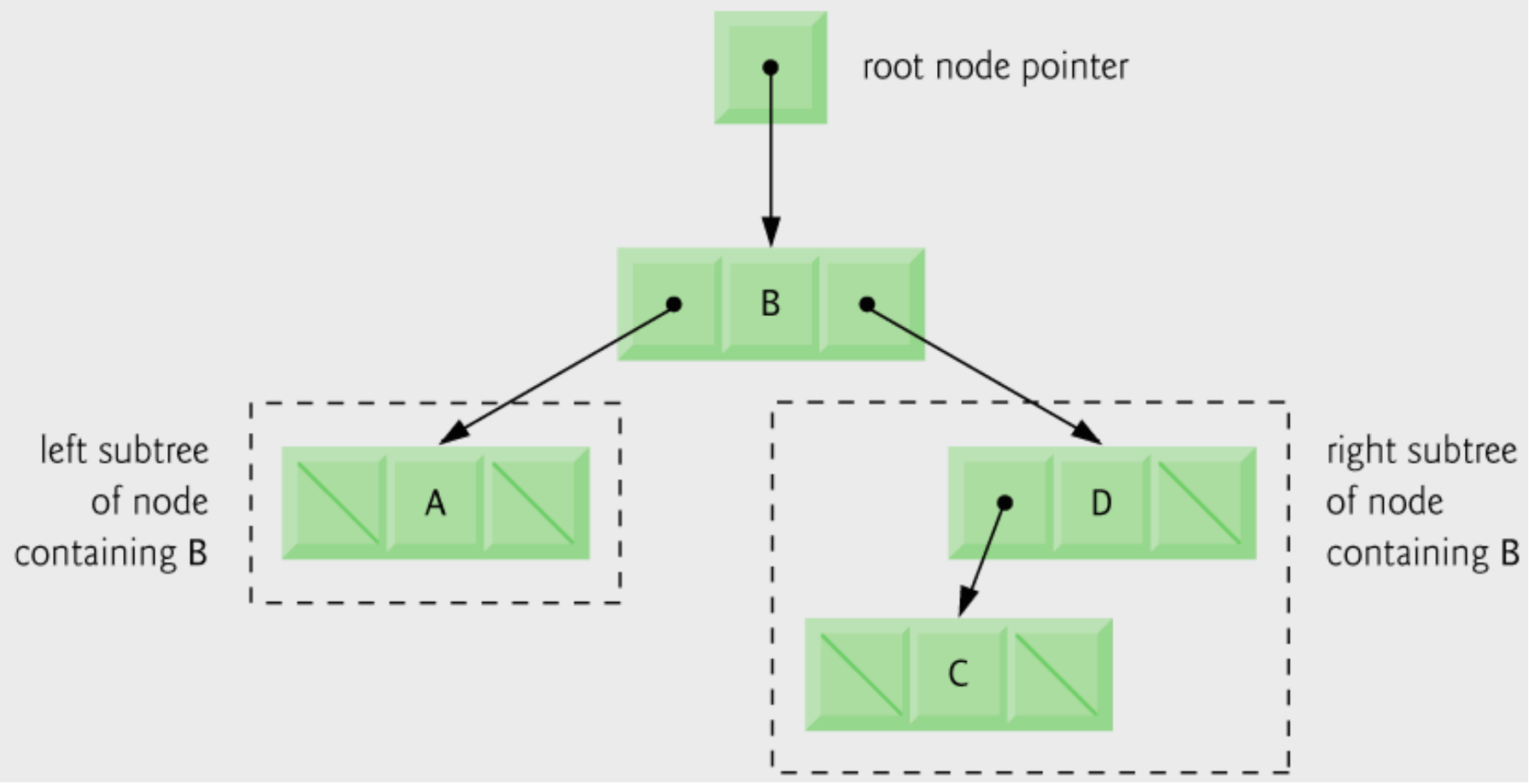
- Profundidad de un árbol binario de n nodos: $\boldsymbol{h = \log_2 n + 1}$

Arboles binarios

- Son aquellos árboles en los que todos los nodos contienen dos enlaces
  - Ninguno, uno, o ambos de los cuales pueden ser NULL
- El **nodo raíz** es el primer nodo en el árbol.
- Cada enlace en el nodo raíz se refiere a un **nodo hijo**.
- Un nodo sin hijos se llama **nodo hoja**.
- Representación en C:

```
struct arbol {
    int  data;
    struct  arbol *izq;
    struct  arbol *der;
};
struct arbol  *raiz = NULL;
```

Arboles binarios – representación gráfica



Recorrido en árboles binarios

- Inorden (SI R SD)
  - Ir hacia el subárbol izquierdo hasta alcanzar la máxima profundidad.
  - Visitar el nodo en curso.
  - Volver hacia el nodo anterior en el árbol y visitarlo.
  - Ir hacia el subárbol derecho del nodo anteriormente visitado siempre que exista y no haya sido visitado previamente, de otra forma, volver hacia el nodo anterior.
  - Repetir los pasos anteriores hasta que todos los nodos hayan sido procesados.

Recorrido en árboles binarios

- Inorden (SI R SD)

```
/* in_orden: imprime el contenido del
  arbol con raiz p en in-orden */
void in_orden(struct arbol *p)
{
  if (p!=NULL) {
     in_orden(p->izq);
     printf("%4d   ",p->data);
     in_orden(p->der);
  }
}
```

Recorrido en árboles binarios

- Preorden (R SI SD)

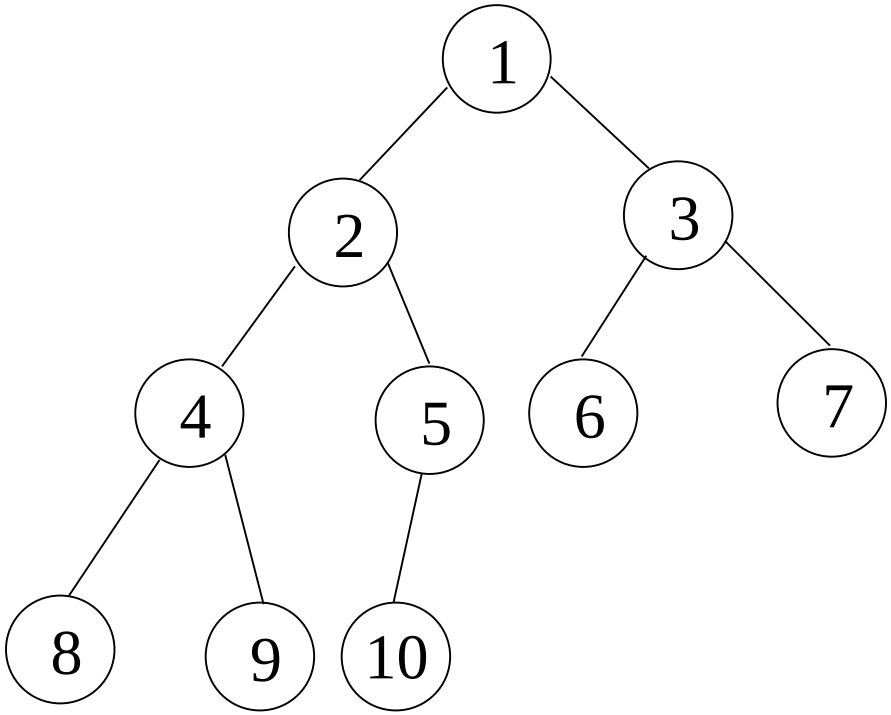
```
/* pre_orden: imprime el contenido del
  arbol con raiz p en pre-orden */
void pre_orden(struct arbol *p)
{
  if (p!=NULL) {
     printf("%4d   ",p->data);
     pre_orden(p->izq);
     pre_orden(p->der);
  }
}
```

Recorrido en árboles binarios

- Postorden (SI SD R)

```
/* post_orden: imprime el contenido
  del arbol con raiz p en post-orden */
void post_orden(struct arbol *p)
{
  if (p!=NULL) {
     post_orden(p->izq);
     post_orden(p->der);
     printf("%4d   ",p->data);
  }
}
```

Recorrido en Arboles



- Inorden: 8 4 9 2 10 5 1 6 3 7
- Preorden: 1 2 4 8 9 5 10 3 6 7
- Postorden: 8 9 4 10 5 2 6 7 3 1

Arbol binario de búsqueda

- Es un árbol en el que el hijo de la izquierda, si existe, de cualquier nodo contiene un valor más pequeño que el nodo padre, y el hijo de la derecha, si existe, contiene un valor más grande que el nodo padre.
- Operaciones:
  - Buscar_Arbol(ValorClave)
  - Insertar(InfoNodo)
  - Suprimir(ValorClave)
  - ImprimeArbol(OrdenRecorrido)

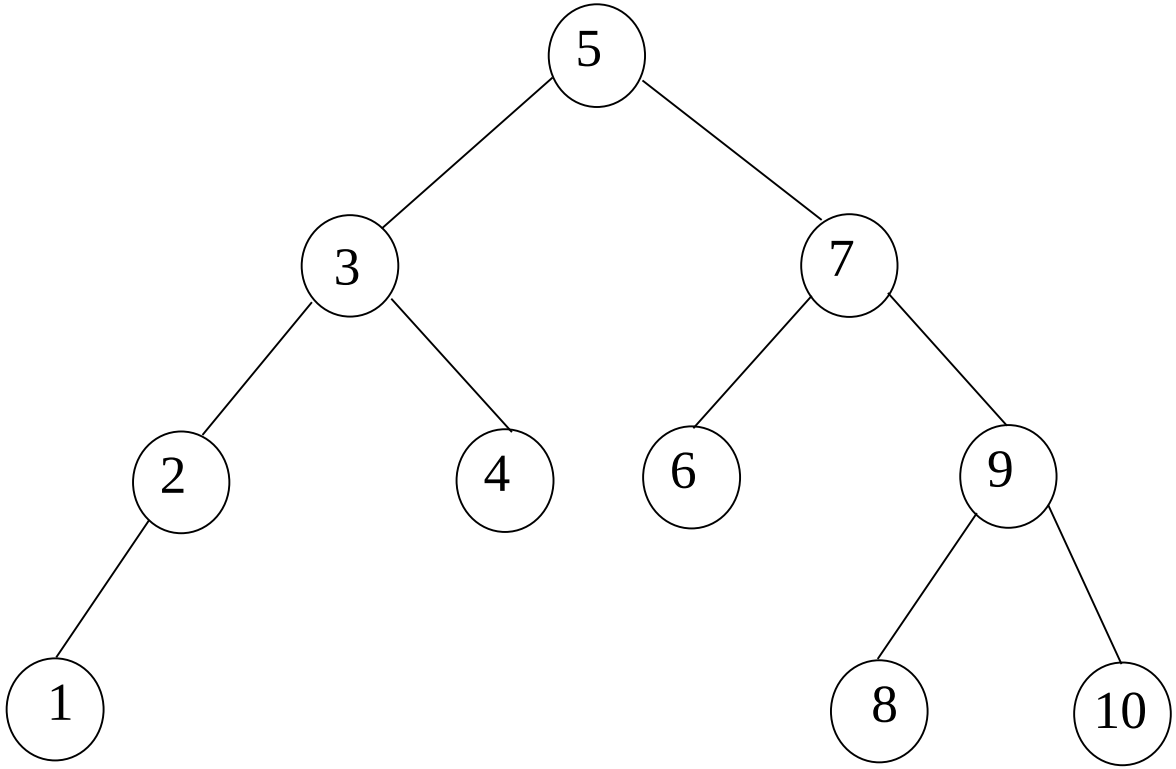
Inserción en un árbol binario de búsqueda

- Sólo se pueden insertar nuevos nodos en los nodos terminales (hojas) o en los nodos internos.
- En un árbol binario de búsqueda se restringe la inserción de nuevos nodos a los nodos terminales debido a la condición de ordenamiento entre los nodos.
- Procedimiento:
  - Se recorre el árbol comparando el nuevo valor con los nodos existentes, dirigiéndose por los descendientes - izquierda o derecha - según el valor a añadir es menor o mayor respectivamente que el nodo en curso.
  - Cuando se llega a un nodo terminal se inserta el nuevo valor como un descendiente de este nodo.

Inserción en un árbol binario de búsqueda

- Construir el árbol binario de búsqueda producido por la siguiente lista:

  5, 3, 4, 7, 2, 6, 9, 1, 8, 10



Inserción en un ABB - código

```
/* addtree: anade un nodo con palabra w. p -> raiz */
struct tnodo *addtree(struct tnodo *p, char *w)
{
 int cond;

 if (p == NULL) {
     p = talloc();
     p->palabra = strdup(w);
     p->cont = 1;
     p->izq = p->der = NULL;
 } else if ((cond = strcmp(w,p->palabra))==0)
     p->cont++;
 else if (cond < 0)
     p->izq= addtree(p->izq, w);
 else
     p->der = addtree(p->der, w);
 return p;
}
```

Inserción en un ABB – código func. aux.

```
#include <stdlib.h>
/* talloc: construye un nodo */
struct tnodo *talloc(void)
{
 return (struct tnodo *) malloc(sizeof(struct tnodo));
}

char *strdup(char *s)
{
 char *p;

 p = (char *) malloc(strlen(s)+1);
 if (p != NULL)
    strcpy(p,s);
 return p;
}
```

Ejemplo de aplicación de un ABB

- Desarrollo de un programa para contar las ocurrencias de las palabras de un texto.
- Como se desconoce la lista de palabras que contiene el texto no puede pensarse en ordenarlas y usar búsqueda binaria.
- Otra alternativa es usar una búsqueda lineal según se lee cada palabra, pero esto tomaría demasiado tiempo (crece cuadráticamente) cuando el número de palabras aumenta.
- La mejor solución, con lo conocido hasta ahora, es tener el conjunto de palabras leídas en cualquier momento ordenadas colocando cada palabra nueva en su lugar. Para ello usaremos un árbol binario de búsqueda.

Frecuencia de ocurrencia de palabras

- El árbol contendrá en cada nodo la siguiente información:
  - Un puntero al texto de la palabra,
  - Un contador del número de ocurrencias,
  - Punteros a los hijos izquierdo y derecho respectivamente.

Frec. de ocurrencia de palabras – código

```
/*********************************************************************\
* Programa: frecuencia_palabras.c                                     *
* Descripción: Prog. que cuenta la frecuencia de ocurrencia de palabras  *
*              de un texto leido desde la linea de comandos           *
*              (redireccionando la entrada)                           *
*              El resultado es una lista ordenada de las palabras junto  *
*              con la frecuencia                                      *
*              El texto debe de constar de letras sin acento y sin signos*
*              de puntuacion                                          *
* Autor: Pedro Corcuera                                               *
* Revisión: 1.0 2/02/2008                                             *
\*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <ctype.h>

#define MAX_CAR 100

struct arb_bin {
   char *palabra;
   int cont;
   struct arb_bin *izq;
   struct arb_bin *der;
};

struct arb_bin *inserta(struct arb_bin *, char *);
void imprime(struct arb_bin *);
```

Frec. de ocurrencia de palabras – código

```
main()
{
 struct arb_bin *raiz;
 char palabra[MAX_CAR];
 raiz=NULL;
 while(scanf(" %s",palabra)!=EOF)
   if (isalpha(palabra[0])) raiz=inserta(raiz,palabra);
 imprime(raiz);
 return 0;
}

struct arb_bin *talloc(void);
char *strdup(char *);

/* inserta: anade un nodo con w, en o bajo p */
struct arb_bin *inserta(struct arb_bin *p, char *w)
{
 int cond;
 if (p==NULL) {
     p = talloc(); p->palabra = strdup(w);
     p->cont = 1; p->izq = p->der = NULL;
 } else if ((cond=strcmp(w,p->palabra))==0)
     p->cont++;
 else if (cond<0) p->izq= inserta(p->izq,w);
 else p->der = inserta(p->der,w);
 return p;
}
```

Frec. de ocurrencia de palabras – código

```
/* imprime: imprime p en in-orden */
void imprime(struct arb_bin *p)
{
 if (p!=NULL) {
    imprime(p->izq);
    printf("%4d  %s\n",p->cont,p->palabra);
    imprime(p->der);
 }
}

/* talloc: construye un nodo del arbol binario */
struct arb_bin *talloc(void)
{
 return (struct arb_bin *) malloc(sizeof(struct arb_bin));
}

/* devuelve un puntero a una copia de una cadena de caracteres */
char *strdup(char *s)
{
 char *p;

 p = (char *) malloc(strlen(s)+1);
 if (p != NULL)
    strcpy(p,s);
 return p;
}
```