AMPLIACIÓN DE FÍSICA DEL ESTADO SÓLIDO

TEMA 1

INTRODUCCIÓN AL LENGUAJE C

CARACTERÍSTICAS GENERALES

El lenguaje C es de nivel medio, potente, versátil y moderadamente moderno

Siempre trabaja con funciones, encerradas entre llaves "{}". Cada línea termina con ";"

Es portable y admite una programación modular

Distingue entre mayúsculas y minúsculas, y tiene una serie de palabras reservadas

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

Admite el uso de librerías preestablecidas

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

LA FUNCIÓN main{}

Programa para convertir temperatura Celsius en Fahrenheit

```
main(){
      double tc, tf, conv;
      double offset = 32.;

      conv = 5./9.;
      tc = (tf - offset)*conv;
}
```

La forma general de la función `main()` es

```
int main(int argv, char* argc){
    Instrucciones, comandos, …
    return 0;
}
```

¿Qué pasa si intentáis compilar el programa de conversión?

INSTRUCCIONES DE PREPROCESADO

Conjunto de instrucciones predefinidas requeridas para compilar y/o ejecutar un programa en C

Se escriben al principio del programa, antes de ninguna instrucción compilable

Instrucción para cargar una libreria

```
#include <librería.h>
```

Instrucción para definir un valor constante

```
#define NOMBRE VALOR
```

```
#include <stdlib.h>
#define OFFSET 32.

main(){
      double tc, tf, conv;

      conv = 5./9.;
      tc = (tf - OFFSET)*conv;
}
```

VARIABLES EN C (I)

Una variable es un nombre simbólico asociado a una cantidad o a una posición de memoria

Tipos de variables en C

Nombre	Tipo	Longitud (bits)
`char`	Carácter	8
`short`	Entero	16
`int`	Entero	32
`long`	Entero	32
`float`	Racional	32
`long long int`	Entero	64
`double`	Racional	64
`long double`	Racional	96

Las variables que se asocian a posiciones de memoria se llaman punteros.

Los nombres de variables no pueden contener ninguno de estos signos:

. , ; : + - = / \ * < > ! ? ~ @ # $ % ^ & | " ` '

VARIABLES EN C (I)

Las variables deben ser declaradas y definidas en un programa.

Declaración de una variable

```
tipo NOMBRE;
```

Definición de una variable

```
NOMBRE = <valor>;
```

```
#include <stdlib.h>
#define OFFSET 32.

main(){
      double tc, tf, conv;

      conv = 5./9.;
      tc = (tf - OFFSET)*conv;
}
```

```
#include <stdlib.h>
#define OFFSET 32.

main(){
      double tf = 23., tc, conv;

      conv = 5./9.;
      tc = (tf - OFFSET)*conv;
}
```

ENTRADA/SALIDA DE DATOS

Funciones de salida

```
printf("Formato de salida", var1, var2,…);
fprintf(fichero, "Formato de salida", var1, var2,…);
```

Funciones de entrada

```
scanf("Formato de entrada", &var1, &var2,…);
fscanf(fichero, "Formato de entrada", &var1, &var2,…);
```

```
#include <stdlib.h>
#include <stdio.h>
#define OFFSET 32.

int main(){
      double tf = 23., tc, conv;

      conv = 5./9.;
      tc = (tf - OFFSET)*conv;
      printf("La temperatura Celsius es %lf\n", tc);

      return 0;
}
```

ENTRADA/SALIDA DE DATOS

Especificadores de formato

Especificador	Variable
%i,%d	int, long
%f,%e,%g	float
%lf,%le,%lg	double
%c	character
%s	string

```
#include <stdlib.h>
#include <stdio.h>
#define OFFSET 32.

int main(void){
     double tf, tc, conv = 5./9.;

     printf("Temperatura Fahrenheit a convertir?\n");
     scanf("%lf", &tf);
     tc = (tf - OFFSET)*conv;
     printf("La temperatura Celsius es %lf", tc);

     return 0;
}
```

VARIABLES EN C (II)

Ámbito de una variable

```
#include <stdio.h>
int main(){
     int a = 137, b = 2;

     {
          int x = 137;
          double a = 3.14;
          printf("Dentro: a = %lf, b = %d, x = %d\n", a, b, x);
     }
     printf("Fuera: a = %d, b = %d, x = %d\n", a, b, x);
     return 0;
}
```

```
#include <stdio.h>
int a = 137;
int main(){
     int b = 2;

     {
          int x = 137;
          double a = 3.14;
          printf("Dentro: a = %lf, b = %d, x = %d\n", a, b, x);
     }
     printf("Fuera: a = %d, b = %d, x = %d\n", a, b);
     return 0;
}
```

OPERADORES EN C

Los operadores pueden ser aritméticos o lógicos

Operadores aritméticos

Símbolo	Operación
+	Suma
–	Resta
*	Producto
/	División
=	Asignación
%	Resto
++	Incremento
--	Decremento

Operadores lógicos

Símbolo	Operación
&&	Y
||	O
==	Igual a
!=	No es igual
>	Mayor que
>=	Mayor o igual que
<	Menor
<=	Menor o igual que

Notación compacta:

```
a = a + b;   →   a += b;
a = a * b;   →   a *= b;
```

Operadores "avanzados"

```
#include <math.h>
```

sqrt(), pow(a,n), sin(), log(), …

OPERADORES EN C

El operador incremento (decremento) puede aparecer antes o después de la variable

`++a` (pre-incremento) → `a` se incrementa *antes* de que se evalúe la expresión donde aparece

`a++` (post-incremento) → `a` se incrementa *después* de que se evalúe la expresión donde aparece

```
a = 2;
b = ++a;
```
a = b = 3

```
a = 2;
b = a++;
```
a = 3; b = 2

El lenguaje C automáticamente promociona las variables

```
double a;
a = 9;
```

```
double a;
a = 5/9.;
```

```
int k;
a = 5/9;
```

```
int k;
k = 5/9.;
```

Las operaciones con variables de distinto tipo se realizan con el operador modo

```
int i = 4;
double a;
a = (double) i/2.;
```

OPERADORES EN C

Jerarquía de los operadores aritméticos


```
int a = 2, b = 3, c = 5;
int d;
```

```
d = a + b*c;
```

17

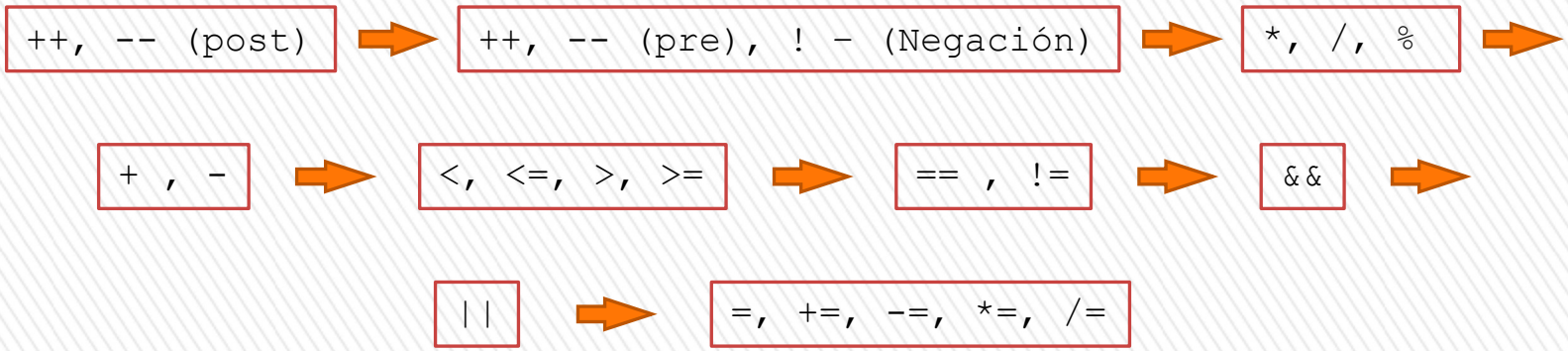
```
d = a + (b*c);
```

17

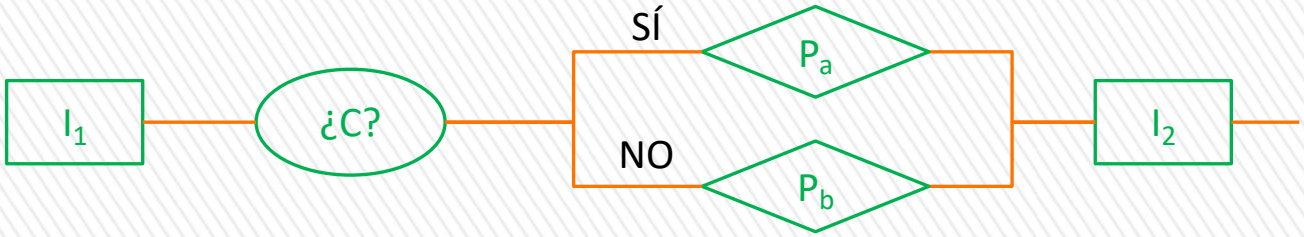
```
d = (a + b)*c;
```

25

Jerarquía de todos los operadores



TOMA DE DECISIONES



Estructura de condicional

```
if ("condición lógica"){
} else {
}
```

```
if ("condición 1"){
} else if ("condición 2"){{
} else {
}
```

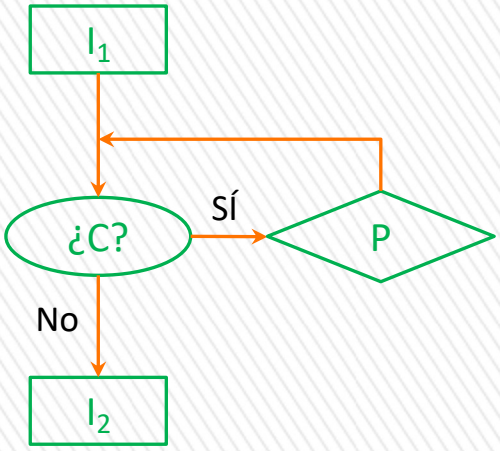
Operador selección

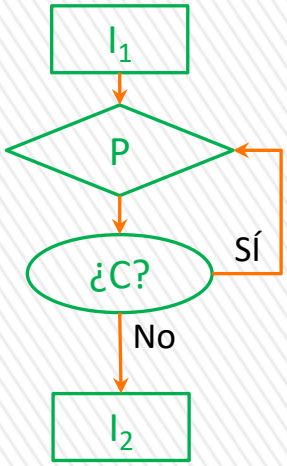
```
(condición) ? (opción a) : (opción b);
```

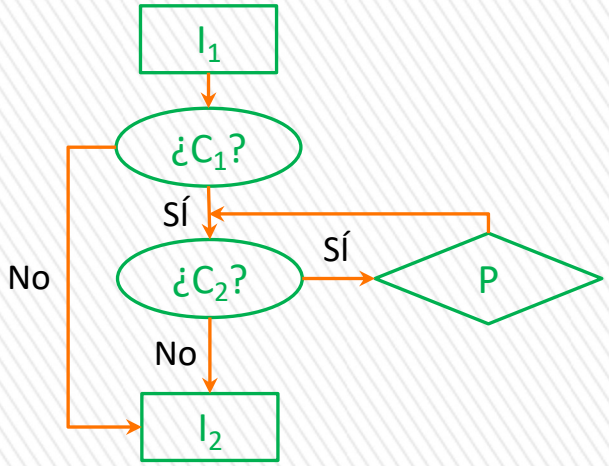
Escribir un programa C que pida un número e indique si el número es par o no.

```
printf ("Introducir un número par\n);
scanf ("%d", &n);
if ((n/2)*2 != n) {
     printf ("Este número no es par\n");
     continue;
} else {…
}
```

CICLOS








```
while(condición){
      procedimiento;
}
```

```
do{
      procedimiento;
} while (condición);
```

```
for (c1; c2; regla) {
procedimiento;
}
```

Escribir un programa C que calcule la suma de los 9 primeros números naturales y muestre el resultado en pantalla, de tres formas distintas.

```
int i = 0, suma = 0;
while(i < 10){
      suma += i++;
}
```

```
int i = 0, suma = 0;
do{
      suma += i++;
} while (i < 10);
```

```
int i, suma = 0;
for(i = 0; i < 10; i++){
      suma += i;
}
```

VECTORES Y MATRICES

Declaración de un vector

```
tipo nombre[longitud];
```

```
double vec[3];
char linea[30];
```

Los elementos de un vector de dimensión N se etiquetan desde 0 hasta $N-1$

La longitud de los vectores debe estar definida cuando se compila el programa

```
int n, data[n];

printf("Dimensión?\n");
scanf("%i", &n);
```

```
#define LEN 20

double data[2*LEN];
```

```
int n = 3, data[n];

printf("Dimensión?\n");
scanf("%i", &n);
```

Existen muchas formas de definir un vector:

```
double x[3]={1.2, 3.5, 1.7};
```

```
double x[]={1.2, 3.5, 1.7};
```

```
double x[3]={1.2};
```

Escribir un programa C que construya los cinco primeros términos de una progresión aritmética.

```
int main(void){
    double vector[5], razon = 0.5;
    int n;

    for(n = 0; n < 5; n++){
        if(n == 0) vector[n] = 1.;
        else vector[n] = razon*vector[n-1];
        printf("Elemento %d = %lf\n", n, vector[n]);
    }
        return 0;
}
```

VECTORES Y MATRICES

Declaración de una matriz

```
tipo nombre[filas][columnas];
```

```
double vec[3][7];
```

En realidad, se trata de `filas*columnas` posiciones de memoria consecutivas, que permiten se tratadas como los elementos de una matriz

Escribir un programa C que calcule el producto de dos matrices.

```
#define F1 5
#define C1 7
#define F2 C1
#define C2 4

int a[F1][C1], b[F2][C2], c[F1][C2];
int i, j, k;
for(i = 0; i < F1; i++){
    for(j = 0; j < C2; j++){
       c[i][j] = 0.;
       for(k = 0; k < C1; k++){
            c[i][j] += a[i][k]*b[k][j];
       }
    }
}
```

FUNCIONES

Declaración de una función

```
tipo nombre(tipo de la lista de parámetros);
```

```
long long int Suma(int);
double Hipot(double, double);
void Ordenar(void);
```

Definición de una función

```
long long int Suma(int n){
    int i = 0, suma = 0;

    while(i <= n){
        suma += i++;
    }
    return suma;
}
```

Los parámetros de una función deben estar definidos cuando se la llama.

```
void ImprimirError(void){
    printf("Error fatal!\n");
}
```

FUNCIONES

Ámbito de una función

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

<declaración de la función A>
<declaración de la función B>

int main(void){
     … Se usan A y B …
}

<definición de la función A>
<definición de la función B>
```

Similar al ámbito de las variables

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

<declaración de la función A>
<declaración de la función B>

int main(void){
     <declaración de C>
}

<definición de la función A>
<definición de la función B>
<definición de la función C>
```

FUNCIONES

Asignación de valores a los parámetros

Los parámetros funcionales son formales

```
#include <stdio.h>

double SumaCuadrados(double, double);

int main(void){
     double x = 4., y = 3., q;
     q = SumaCuadrados(x, y);
     printf("Primera suma: %lf\n", q);
     q = SumaCuadrados(5, 2*2);
     printf("Segunda suma: %lf\n", q);
}

double SumaCuadrados(double a, double b){
     return (a*a + b*b);
}
```

FUNCIONES

Asignación de valores a los parámetros

Los parámetros funcionales son formales

```
#include <stdio.h>

double RestaCuadrados(double, double);

int main(void){
     double x = 4., y = 3., q;
     q = RestaCuadrados(x, y);
     printf("Primera suma: %lf\n", q);
     q = RestaCuadrados(y, x);
}

double RestaCuadrados(double a, double b){
     return (a*a - b*b);
}
```

FUNCIONES

Una función no puede cambiar el valor de sus parámetros de entrada

```
#include <stdio.h>

void Intercambio(double, double);

int main(void){
     double x = 4., y = 3.;
     printf("Antes: x = %lf, y = %lf\n", x, y);
     Intercambio(x, y);
     printf("Después: x = %lf, y = %lf\n", x, y);
}

void Intercambio(double a, double b){
     double temp;

     temp = a;
     a = b;
     b = temp;
}
```

Los elementos de un vector no se pueden usar (directamente) como parámetros de entrada de una función

PUNTEROS

Un puntero es una variable que almacena la posición de memoria asignada a otra variable

Declaración

```
tipo *nombre;
```

```
char *apellido;
int *dni;
double *volumen;
```

Definición

```
nombre = &variable;
```

```
dni = &numero;
volumen = &vol;
```

```
#include <stdio.h>

int main(void){
     double a, *pd, *pe;
     pd = &a;
     pe = pd;
     a = 3.14;
     printf("Punteros pd=%p, pe=%p, variable a=%lf\n", pd,pe,a);
     a = 2.73;
     printf("Punteros pd=%p, pe=%p, variable a=%lf\n", pd,pe,a);
}
```

Un puntero permite acceder al valor de la variable en la dirección a la que apunta

```
double a = 3.14, b, *pd;
pd = &a;
b = *pd;
```

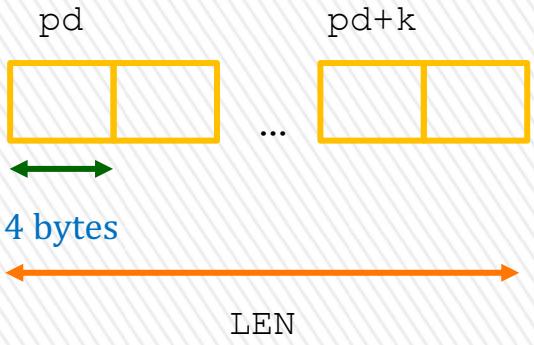
PUNTEROS

Los punteros permiten tratar los vectores y matrices de forma sencilla

```
int datos[10], n, *pi;
pi = &data[0]                ¿¿pi + 1??      ¿¿*(pi + 1)??
n = *pi;
```

```
#include <stdio.h>
#define LEN 5

int main(void){
     float data[LEN], *pd;
     int i;
     pd = &data[0];
     for(i = 0; i < LEN; i++){
        printf("&data[%d] = %p, pd + %d = %p\n",
          i, &data[i], i, pd + i);
     }
}
```



Un vector o matriz sin [] equivale a un puntero (del mismo tipo) a su primer elemento

```
double data[10], *pd;
int n[5], *pi;
pd = data;
pi = n;
```

PUNTEROS

Aritmética de punteros

No todos los operadores aritméticos actúan sobre punteros en C

```
#include <stdio.h>

int main(void){
      double data[10] = {3.14, 1.57}, *pd;
      double a, b, c;
      int k = 2;

      pd = data;
      a = *(pd + 1);
      pd++;
      b = *(pd - 1);
      c = *(pd + k);
      printf("a = %lf, b = %lf, c = %lf\n", a, b, c);
      return 0;
}
```

Prioridad de los operadores sobre punteros

```
int k[2], *pi;
k[0] = 137;
pi = k;
```

¿*pi++;?

Se evalúa el contenido de `pi`, y luego se incrementa `pi` en una unidad

¿(*pi)++;?

Se incrementa en una unidad el valor almacenado en `pi`

PUNTEROS

Asignación dinámica de memoria

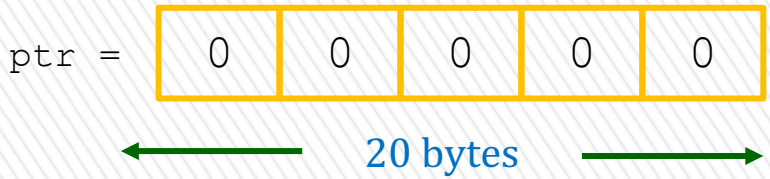
Variable vectorial (o matriz) definida como puntero.

(type*)malloc(n*sizeof(type));

```
int *ptr;
ptr = (int*)malloc(5*sizeof(int));
```

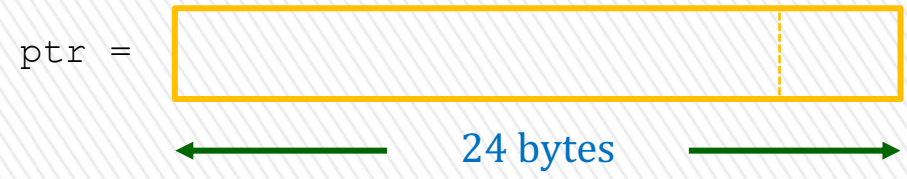
(type*)calloc(n,sizeof(type));

```
int *ptr;
ptr = (int*)calloc(5,sizeof(int));
```



realloc(pointer, newSize);

```
int *ptr;
ptr = realloc(ptr,6*sizeof(int));
```



PUNTEROS

Asignación dinámica de memoria

Comprobación de asignación correcta

```
#include <stdio.h>

int main(void){
      double *data;
      int k = 2;

      data = (double*)calloc(k, sizeof(double));
      if(data == NULL) {
            printf("Error de asignación a data.\n");
            exit (0);
      } else {
      }
      …
}
```

Liberación dinámica de memoria

```
free(pointer);
```

PUNTEROS

Asignación dinámica de memoria

Asignación de matrices

```
#include <stdio.h>

int main(void){
     float **matriz;
     int nf, nc;
     int i;

     matriz = (float**)calloc(nf, sizeof(float*));
     for(i = 0; i < nf; i++)
        matriz[i] = (float*)calloc(nc, sizeof(float));

     if(matriz == NULL){
          printf("Error de asignación de memoria\n");
          exit(0);
     }
     …
}
```

PUNTEROS

Asignación dinámica de memoria

Asignación de matrices

```
#include <stdio.h>

int main(void){
     float *matriz[nf];
     int nf, nc;
     int i;

     for(i = 0; i < nf; i++)
        matriz[i] = (float*)calloc(nc, sizeof(float));

     if(matriz == NULL){
          printf("Error de asignación de memoria\n");
          exit(0);
     }
     …
}
```

PUNTEROS

Punteros como parámetros de funciones

```
void Intercambio(double, double);

void Intercambio(double a, double b){
     double temp;

     temp = a;
     a = b;
     b = temp;
}
```

```
void Intercambio(double*, double*);

void Intercambio(double *a, double *b){
     double temp;

     temp = *a;
     *a = *b;
     *b = temp;
}
```

```
#include <stdio.h>

int main(void){
     double x = 4., y = 3.;
     printf("Antes: x = %lf, y = %lf\n", x, y);
     Intercambio(&x, &y);
     printf("Después: x = %lf, y = %lf\n", x, y);
}
```

PUNTEROS

Entrada / salida con archivos

```
fprintf(fichero, "Salida", var1, var2,…);
fscanf(fichero, "Entrada", &var1, &var2,…);
```

El tratamiento de archivos se realiza a través de un puntero a archivo

```
FILE *fp;
fp = fopen("Nombre", "modo");
```

Modo	Acción
"r"	Sólo lectura
"w"	Sólo escritura
"a"	Sólo actualización
"r+", "w+"	Lectura y escritura

```
#include <stdio.h>

int main(void){
      int i;
      FILE *archivo;
      archivo = fopen("Datos.dat", "w+");

      for(i = 0; i < 10; i++) fprintf(archivo, "%d %d\n", i, i*i);
      fclose(archivo);
      return 0;
}
```

PUNTEROS

Entrada / salida con archivos

Escribir un programa C que escriba en un fichero los diez primeros números naturales y sus cuadrados, y que luego identifique cuáles de ellos son múltiplos de 3.

```
#include <stdio.h>

int main(void){
      int i, j, k;
      FILE *archivo;
      archivo = fopen("Datos.dat", "w+");

      for(i = 0; i < 10; i++) fprintf(archivo, "%d %d\n", i, i*i);

      rewind(archivo);

      for(i = 0; i < 10; i++){
          fscanf(archivo, "%d %d", &j, &k);
          if ((k/3)*3 == k && k != 0){
              printf("El numero %d es divisible por 3\n", k);
          }
      }
      fclose(archivo);
      return 0;
}
```

PROGRAMACIÓN MODULAR

El lenguaje C permite distribuir fragmentos autónomos de código (o librerías) en archivos distintos

Estructura general de un programa en C

```
#include <stdlib.h>
#include <stdio.h>
#include "parametros.h"
#define N 20

type funcion#1(args1);
        …
type funcion#N(argsN);


int main(void){
        …
   return 0;
}
```

Contenido del directorio

main.c
parametros.h
fragmento#1.c
fragmento#2.c
…
fragmento#N.c

Declaracion.c

PROGRAMACIÓN MODULAR

Archivo parametros.h

```
#include <stdlib.h>
#include <stdio.h>
…
#define A 100.
#define N 20
…

extern type variable#1;
extern type variable#2;
…
extern type variable#N;
```

Archivo declaracion.c

```
#include "parametros.h"

type variable#1;
type variable#2;
…
type variable#N;
```

Archivo fragmento#k.c

```
#include "parametros.h"


type funcion#k(argsk){
…
}
type funcion#k+1(argsk+1){
…
}
…
```

En cada archivo fragmento#k.c se incluyen funciones relacionadas por su acción en el programa global

PROGRAMACIÓN MODULAR

Código de Dinámica Molecular para difusión en estado sólido

def_functions.h

params.h

aleat.c

declaration.c

dynamics.c

main.c

output.c

properties.c

setupJob.c

singleStep.c

structure.c

PROGRAMACIÓN MODULAR

def_functions.h

```
#define AllocMem(a, n, t)     \
         a = (t *) malloc ((n)*sizeof(t))

#define VAdd(v1,v2,v3)                    \
   v1.x = v2.x + v3.x, v1.y = v2.y + v3.y, v1.z = v2.z + v3.z
#define VSub(v1,v2,v3)  VAdd(v1,v2,-v3)
#define VDot(v1,v2)           \
   v1.x * v2.x + v1.y * v2.y + v1.z * v2.z
#define VSAdd(v1, v2, s3, v3) \
   v1.x = v2.x + s3*v3.x,     v1.y=v2.y+s3*v3.y, v1.z=v2.z+s3*v3.z
#define VSet(v, sx, sy, sz)   \
   v.x = sx, v.y = sy, v.z = sz
#define VSetAll(v,s)  VSet(v,s,s,s)
#define VZero(v)    VSetAll(v,0)
#define VVSet(v,w)                        \
   v.x = w.x, v.y = w.y, v.z = w.z
#define VVSAdd(v1,s2,v2)   VSAdd(v1, v1, s2, v2)
#define VLengSq(v)  VDot(v,v)
#define VNorm(v)   sqrt(VLengSq(v))

#define VWrap(v,L,t)          \
   if (v.t >= 0.5 * L.t) v.t -= L.t;    \
   else if (v.t < -0.5*L.t) v.t += L.t
#define VWrapAll(v,L)                     \
   {VWrap(v,L,x); VWrap(v,L,y); VWrap(v,L,z);}
#define do_part for (i=1; i<=npart; i++)
```

PROGRAMACIÓN MODULAR

params.h

```
#include <math.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <time.h>
#include "def_functions.h"

#define LR 102133*10
#define npart 50
#define m_1 1.
#define m_2 1.5
#define sigma_1 1.
…
typedef struct {double x,y,z;} vector;
typedef struct {double val,sum,sum2;} prop;
typedef struct {double a, b;} TypePart;

extern prop PotEnergy;
extern prop KinEnergy;
extern prop Energy;
extern prop Temp;
extern prop Virial;
…
```

```
extern int i, j, k, l;
extern int cfuerzas, countRdf, countAcfAv;
extern int iaccum, iff, it
extern int MoreCycles;
…

extern double al;
extern double energy;
…

extern double histRdf[sizeHistRdf];
extern double mass[1+npart];
extern double diameter[1+npart];

extern void AccumProps (int iaccum);
extern void AccumVacf();
extern void AllocArrays();
extern void ApplyBoundaryCond();
extern void ComputeForces();
extern void InitCoords();
extern void InitForce();
extern void InitProps();
extern void InitVacf();
extern void InitVels();
…
```

PROGRAMACIÓN MODULAR

declaration.c

```
#include "params.h"

int i, j, k, l;
int cfuerzas, countRdf, countAcfAv;
int iaccum, iff, it;
int MoreCycles;
int xi;

double *avAcfVel;
double diameter[1+npart];
double energy;
double histRdf[sizeHistRdf];
double intAcfVel_1, intAcfVel_2;
double ir2,ir6;
double latticeCorr, ljr;
double mass[1+npart];
double m2;
double rcut, rij2;
double sigmaij;
double t;
double v2;
```

```
vector dr;
vector f[1+npart];
vector Lv = {L, L, L};
vector LvR = {L-0.5, L-0.5, L-0.5};
vector r[1+npart];
vector v[1+npart];

prop KinEnergy;
prop Energy;
prop PotEnergy, Pressure;
prop SpecificHeat;
prop Temp;
prop Virial;

TBuf *tBuf;

TypePart *avAcfVel;
TypePart intAcfVel;
```

PROGRAMACIÓN MODULAR

main.c

```
#include "params.h"


int main(){

      SetupJob();

      do{
          SingleStep();
      } while (it < nt);

      EvalProps(2);
      Output();

      return 0;
}
```

PROGRAMACIÓN MODULAR

setupJob.c

```
#include "params.h"

void SetupJob(){
    it = 0;
    AllocArrays();
    InitVacf();
    InitProps();
    InitCoords();
    InitVels();
    countRdf = 0;
    ComputeForces();
    AccumProps(0);
    EvalProps(0);
    MeasureTemp(v);
    Messages();
    AccumProps(1);
}
```

```
void AllocArrays(){

    int nb;
    AllocMem (avAcfVel, nValAcf, TypePart);
    AllocMem (tBuf, nBuffAcf, TBuf);
    for (nb = 0; nb < nBuffAcf; nb ++){
        AllocMem (tBuf[nb].acfVel, nValAcf,
TypePart);
        AllocMem (tBuf[nb].orgVel, npart, vector);
    }
}

void InitProps(){
    for (i=1; i<=npart; i++) {
        if (i <= n_1) {
            mass[i] = m_1;
            diameter[i] = sigma_1;
        } else {
            mass[i] = m_2;
            diameter[i] = sigma_2;
        }
    }
}

void InitCoords(){
}
```

PROGRAMACIÓN MODULAR

singleStep.c

```
#include "params.h"


void SingleStep(){
     ++it;
     t = it*dt;
     LeapfrogStep(1);
     ApplyBoundaryCond();
     ComputeForces();
     LeapfrogStep(2);
     MeasureTemp(v);
     MeasureMom(v);
     AccumProps(1);
     if (it/(StepOutput*1.) == it/StepOutput){
          AccumProps(2);
          EvalProps(1);
          Messages();          // $ imprime en pantalla información de
la evolución del sistema
//        Output(it/StepOutput);
          AccumProps(0);
     }
     if (it >= StepEquil && (it - StepEquil)%stepRdf==0) EvalRdf();
     if (it >= StepEquil && (it - StepEquil)%stepAcf==0) EvalVacf();
}
```