Programación : repaso de C (4)

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Agosto 2009

Outline

Outline

Funciones

- Son pedazos de código que se pueden llamar unos entre ellos (incluso recursivamente).
- Pueden regresar un valor o no (si el tipo de regreso es *void*).
- Un programa tiene que tener una función *main*, que no puede estar llamada.
- La función empieza por declaraciones de los variables, luego viene el cuerpo de las instrucciones y la instrucción *return* que regresa el resultado (menos estricto en C99).

Típicamente :

```
tipoX funcion(tipoY arg1, tipoZ arg2,...) {
        tipoX t;
        ...
        return t;
}
```

Funciones : ejemplo

```
int f(unsigned int n) {
        int result = 1;
        int k;
        for (k=1;k<=n;k++)
                result *= k;
        return result;
}
```

Funciones

- Llamar a la función se hace con el nombre de la función, y con los parámetros dentro de los () (variables o constantes).
- No es estándar el orden de evaluación de los parámetros (cuidado con ++...).

```
int result = f(15);
int a=5;
int b=f(a);
```

Funciones : void

A una función que no regresa nada, se le aplica el tipo *void* (tipo nulo). En este caso la palabra llave return se puede usar (sin argumentos) o no.

```
void g(unsigned int n) {
        printf("%d \n",f(n));
}
```

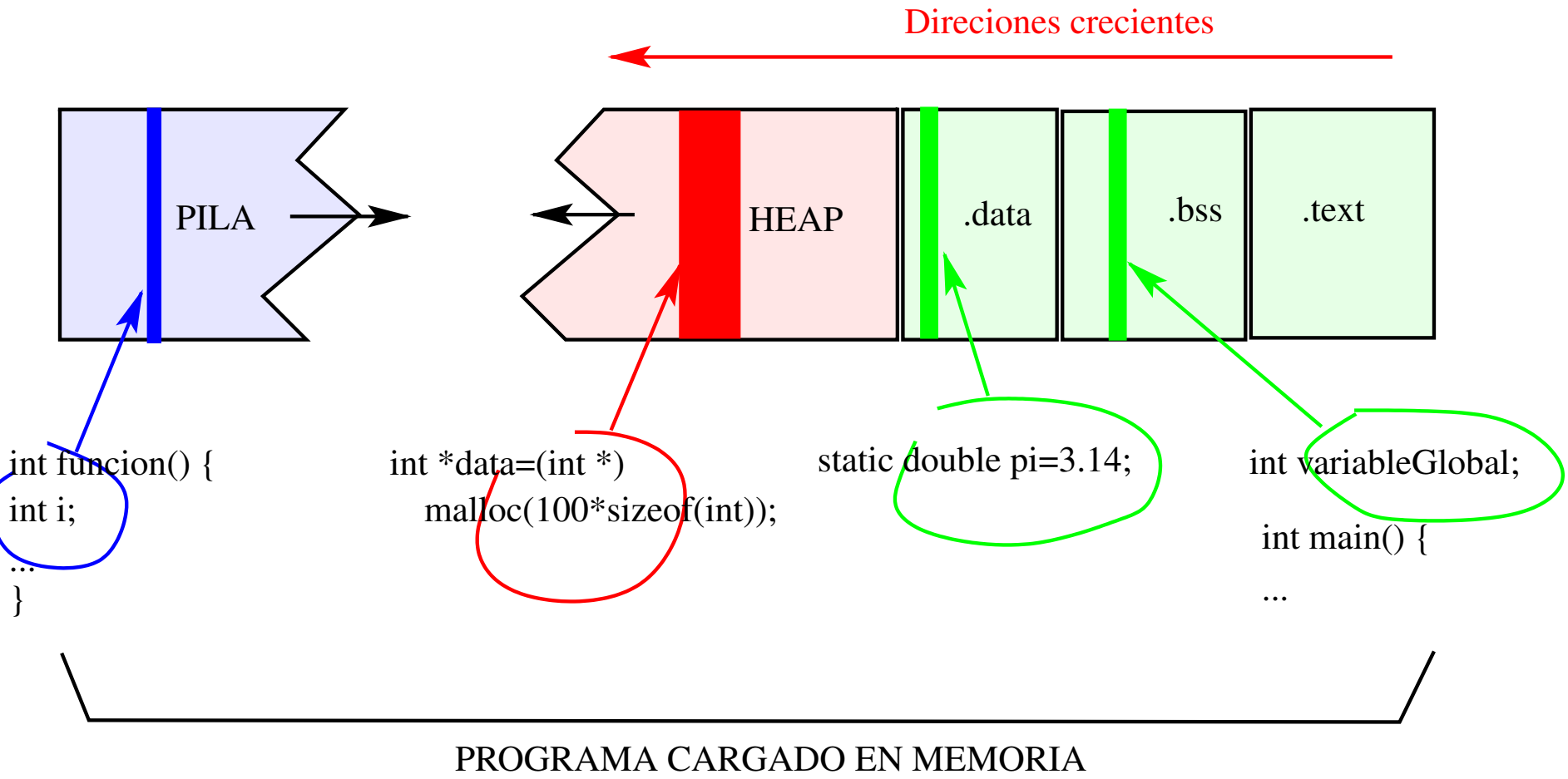
Funciones : ejemplo recursivo

```
int f(unsigned int n) {
        if (n==0)
                return 1;
        else
                return n*f(n-1);
}
```

Funciones : memoria

Los recursos en memoria dentro de cada función son locales : van a desaparecer cuando salimos de la función. Por eso la estructura de pila para la memoria : cada nueva llamada pide un espacio de memoria que se consigue abajo del espacio memoria de la función llamante; cuando se sale, este espacio está liberado

Funciones : memoria

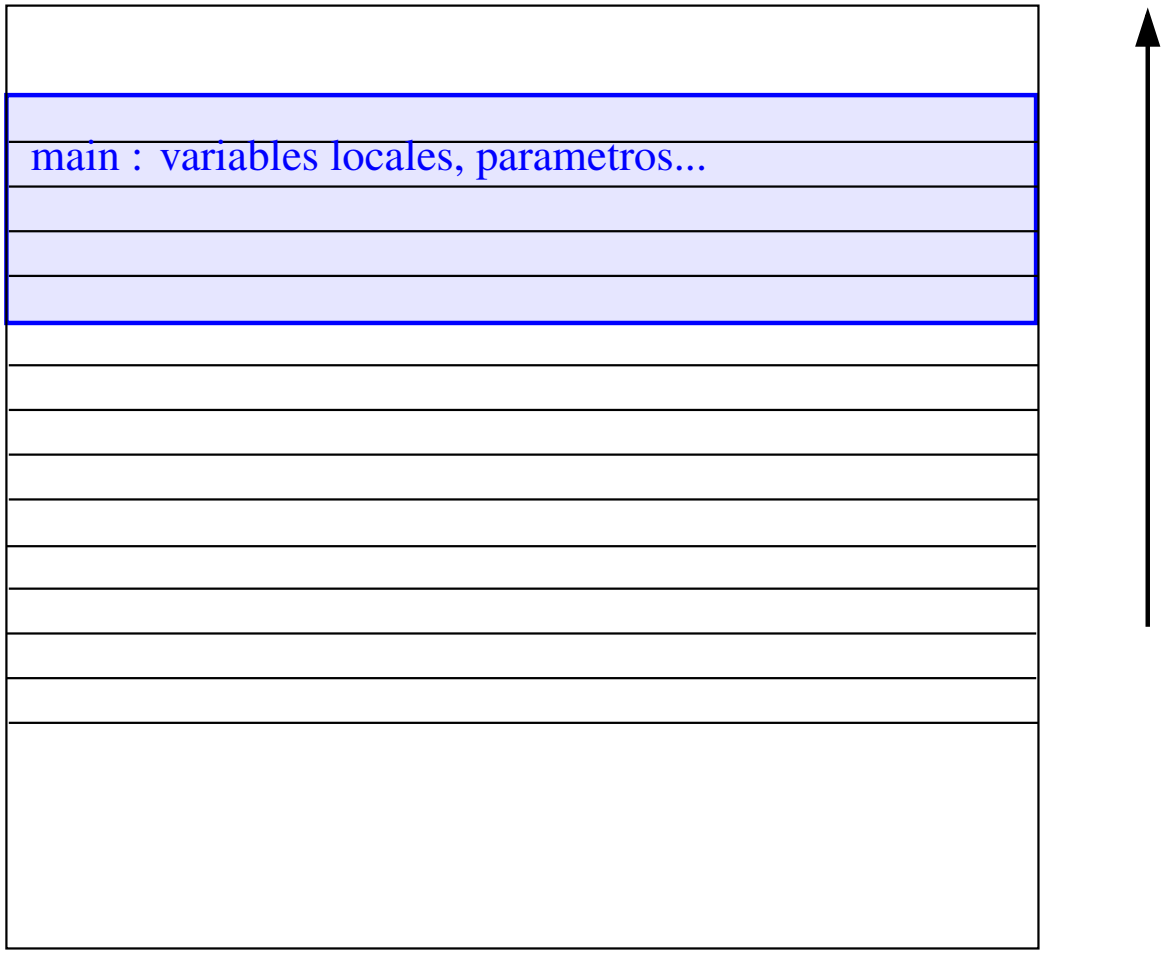


Funciones : memoria

```
int main() {
int a;
int b;
int c = norm(a,b);
return 0;
}

int norm(int x,int y) {
int n = prod(x,x)+prod(y,y);
return n;
}

int prod(int x,int y) {
int p=x*y;
return p
}
```

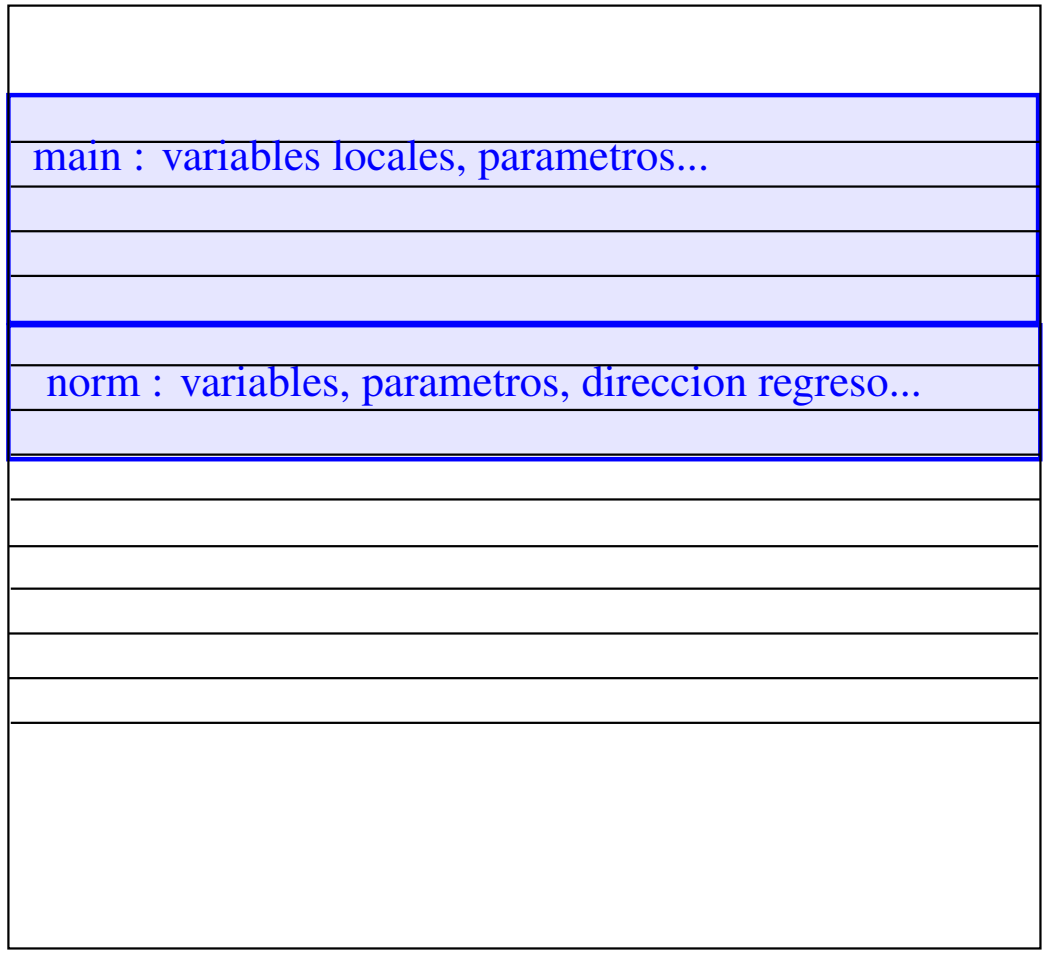


Funciones : memoria

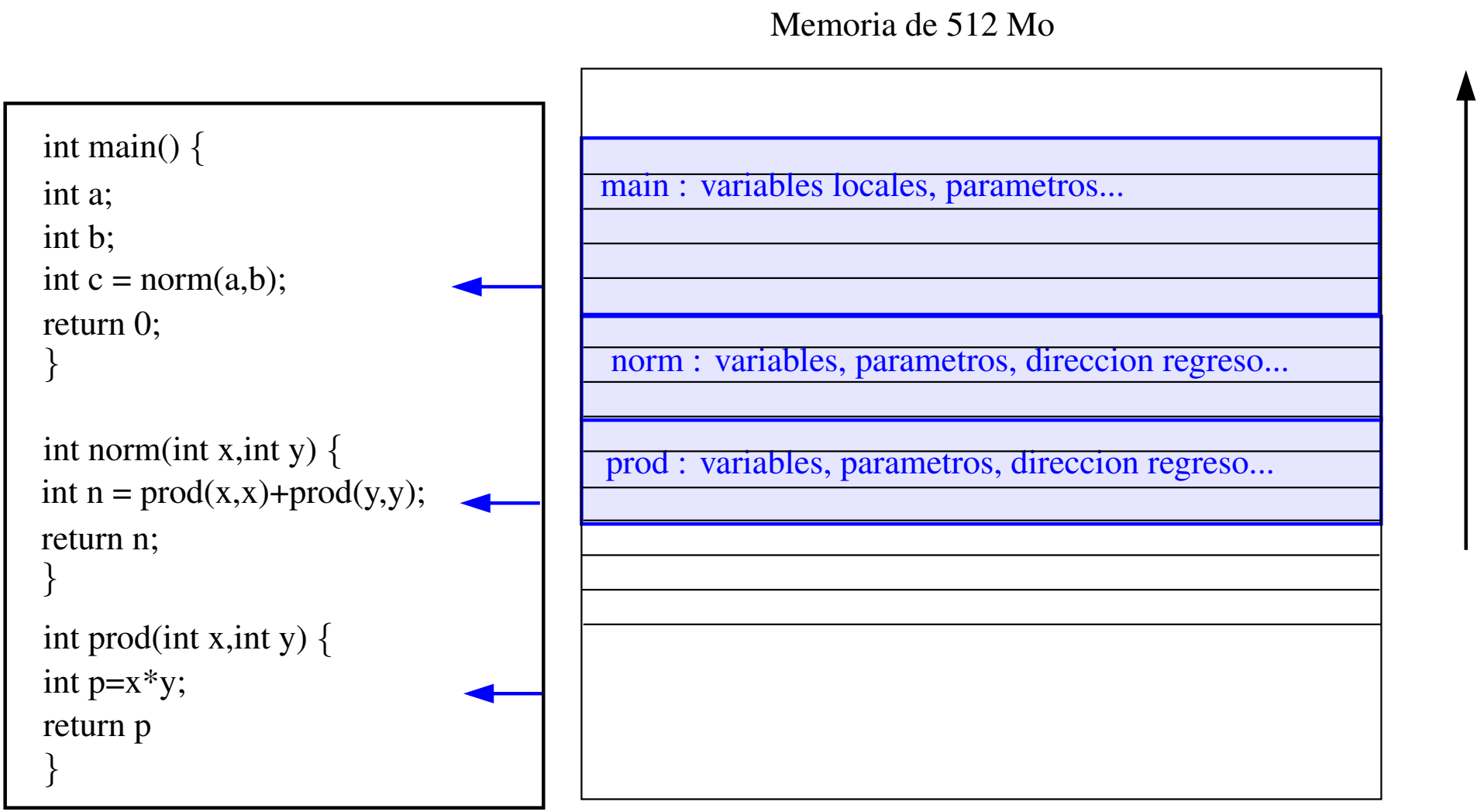
```
int main() {
int a;
int b;
int c = norm(a,b);
return 0;
}

int norm(int x,int y) {
int n = prod(x,x)+prod(y,y);
return n;
}

int prod(int x,int y) {
int p=x*y;
return p
}
```



Funciones : memoria

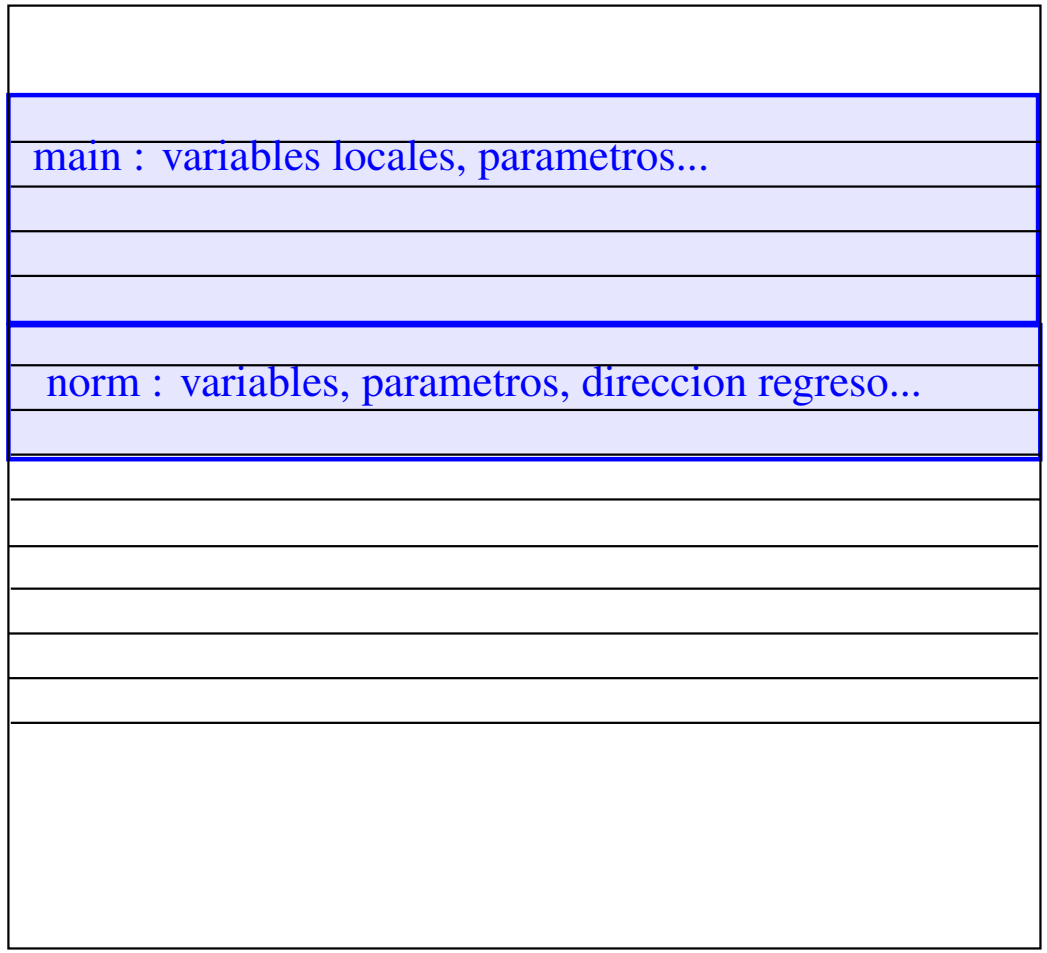


Funciones : memoria

```
int main() {
int a;
int b;
int c = norm(a,b);
return 0;
}

int norm(int x,int y) {
int n = prod(x,x)+prod(y,y);
return n;
}

int prod(int x,int y) {
int p=x*y;
return p
}
```

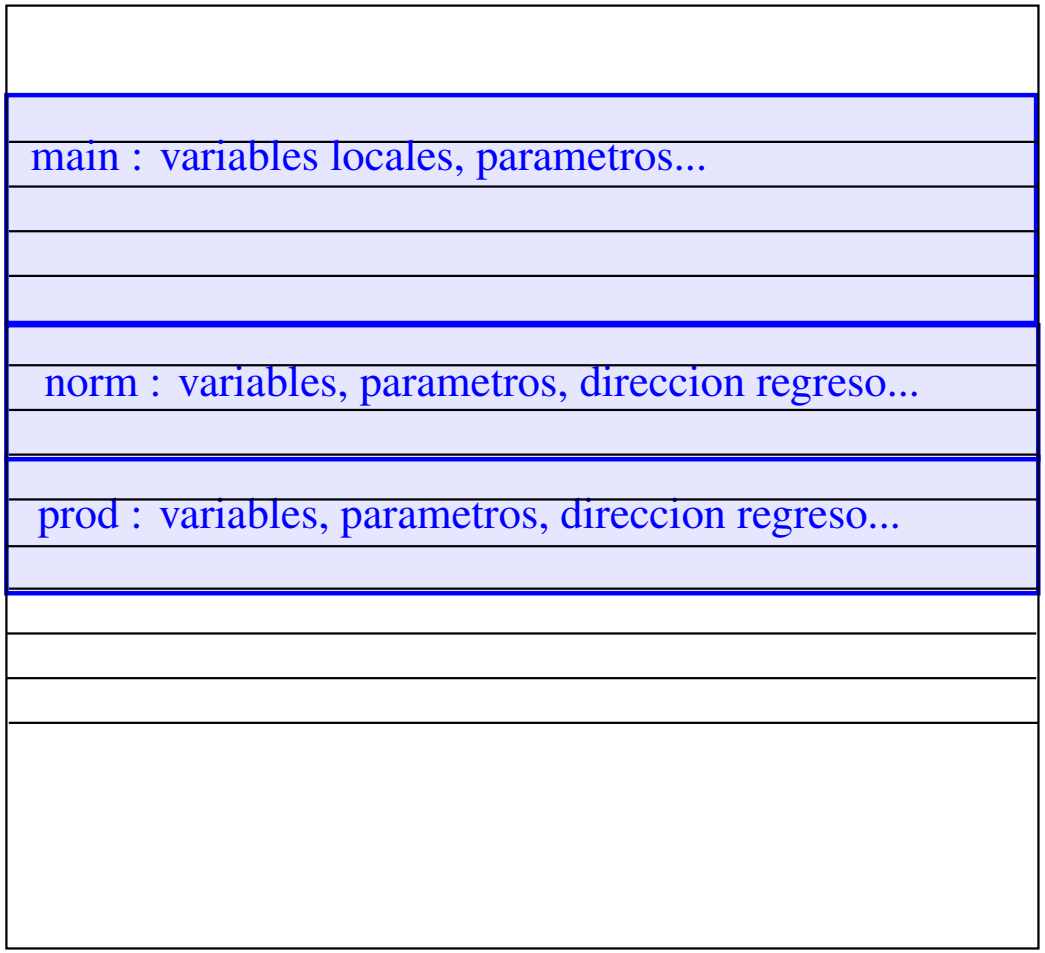


Funciones : memoria

```
int main() {
int a;
int b;
int c = norm(a,b);
return 0;
}

int norm(int x,int y) {
int n = prod(x,x)+prod(y,y);
return n;
}

int prod(int x,int y) {
int p=x*y;
return p
}
```

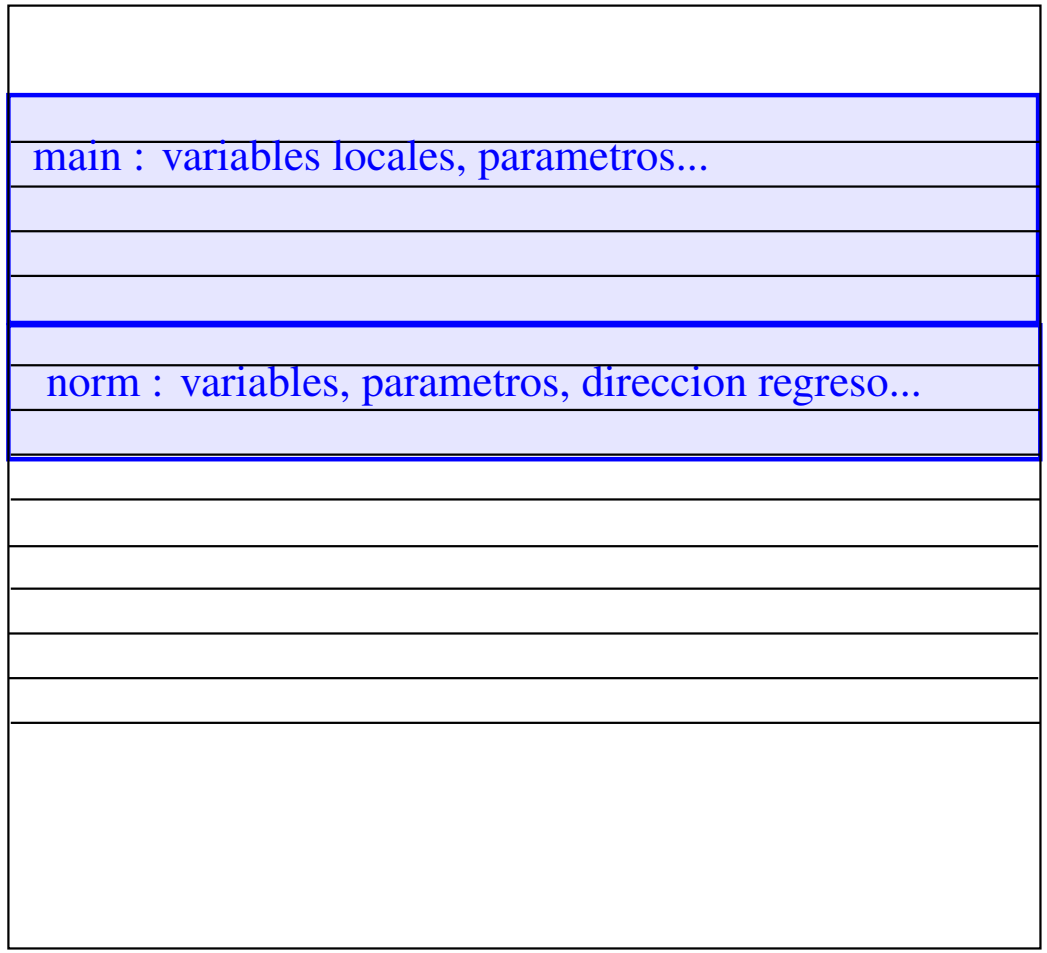


Funciones : memoria

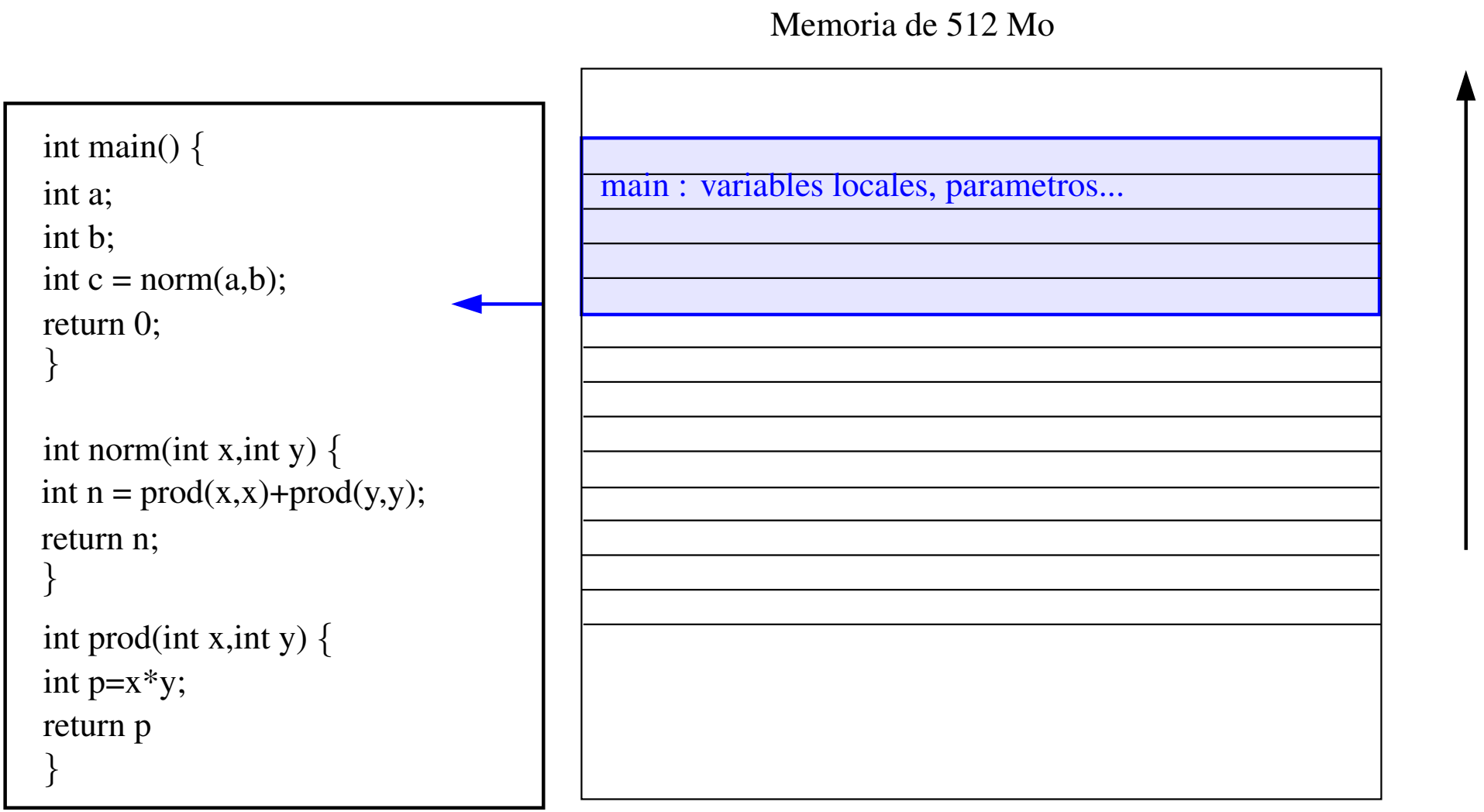
```
int main() {
int a;
int b;
int c = norm(a,b);
return 0;
}

int norm(int x,int y) {
int n = prod(x,x)+prod(y,y);
return n;
}

int prod(int x,int y) {
int p=x*y;
return p
}
```



Funciones : memoria



Funciones : recursividad

Dado este mecanismo de pila, uno entiende que usar funciones recursivas, aunque puede escribirse en un código muy claro, puede ser muy costoso en términos de memoria !

Funciones : variables locales

El caso por default es que las variables declaradas en las funciones están reservadas en la pila y que sólo existen el tiempo de estar en la función. Son puramente locales. El espacio es limitado, generalmente a unos Mo.

```
Bidarray[CLASE3][18:48]>ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) 6144
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 256
pipe size            (512 bytes, -p) 1
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 266
virtual memory          (kbytes, -v) unlimited
```

Funciones : variables estáticas

Unas variables pueden estar guardadas diferentemente del mecanismo de pila : quedan de visibilidad local pero su espacio memoria esta guardado (el valor no se pierde entre dos llamadas a la función). Se introducen por la palabra llave static.

```
int m(unsigned int n) {
        static int k=1000; // .data
        static int l;      // .bss
        int c=10;          // pila
        ...
}
```

Funciones : variables globales

Hay también variables que son definidas fuera de toda función (en el archivo de código), se llaman variables globales. Estarán visibles en todo código que sigue su declaración.

```
int nGlobal;
void f() {
        print("Valor de nGlobal %d\n",nGlobal++);
        return;
}
int main() {
        int k;
        nGlobal=19;
        for (k=0;k<10;k++)
                f();
        return 0;
}
```

Funciones : variables globales

Cuidado con esas variables, por ejemplo por conflicto de nombre :

```
int n;
void f() {
        int n=0;
        n=18;
        ...
        return;
}
```

Funciones

- No se puede definir una función dentro de otra, pero solo antes o después de otra.
- Una función que llama otra debe de saber antes de la llamada que esa existe : una declaración de la función es indispensable si es definida después de la que llama (el prototipo solo : tipo, nombre, argumentos)

```
int  f(unsigned int n);  // Declaration de f

void g(unsigned int n) { // Definicion de g
        printf("%d\n",f(n));
}
int f(unsigned int n) {  // Definicion de f
        int p=(n==0)?1:n*f(n-1);
        return p;
}
```

Funciones

Para no tener líos con el orden de las funciones (declaraciones, definiciones), se suele poner todas las declaraciones en un archivo *header* separado del en que vienen las definiciones, con una extensión .h.

Funciones : headers

Contenido de func.h :

```
int  f(unsigned int n);  // Declaración de f
void g(unsigned int n); // Declaración de g
```

Funciones : headers

Contenido de func.c, que incluye el contenido de func.h (comanda al preprocesador) :

```
#include "func.h"
void g(unsigned int n) { // Definición de g
...
}
int  f(unsigned int n) {   // Definición de f
...
}
```

Funciones : cast de parámetros

Otra ventaja de las declaraciones es que permite prever si se necesita hacer conversiones de tipos, cuando es necesario.

```
#include <stdio.h>

//void h(int m,int n);

int main() {
   float b=6,c=9;
   h(b,c);
}
void h(int m,int n) { // Definicion de g
   printf("%d %d\n",m,n);
}
```

Funciones externas

Se puede usar funciones que nosotros no definemos, pero que tomamos de otras librerías; en este caso se tiene que declararla, y se hace con la palabra llave extern :

```
extern int putchar(int);
```

Funciones : parámetros

- Los parámetros están procesados exactamente como variables locales : a la llamada de la función, están copiados en el segmento memoria de la pila correspondiendo a la función.
- Por eso al salir de la función, esas copias de trabajo no existen mas !
- El comportamiento global es que el valor de los parámetros no puede estar cambiado dentro de una función.

```
void h(double x) {
        x=1.0;
        return;
};
double y = 3.0;
h(y);
```

Funciones : parámetros

Para cambiar el valor de parámetros, hay que pasarle a la función el apuntador hacia este valor; es útil cuando se necesita cambiar varias cosas de regreso

```
void h(double *x) {
        *x=1.0;
};
double y = 3.0;
h(&y);
```

Funciones : parámetros

Una cosa comun es usar el *return* para regresar una error, y hacer todos los otros regresos por apuntadores

```
int h(double *x) {
        *x=1.0;
        ...
        if (problem)
                return -1;
        return 0;
};
double y = 3.0;
int err = h(&y);
```

Funciones : main

El entero de regreso de main esta enviado al sistema. 0 (EXIT_SUCCESS) significa que todo paso bien, valores no nulas a ejecuciones problematicas (como EXIT_FAILURE). Alternativamente se puede usar exit(int status);

Funciones : main

Hasta ahora vimos como prototipo valido para *main*

```
int main ();
```

pero hay otro :

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

Funciones : main

argc es el numero de parametros pasados al programa al lanzarlo (contando el nombre del programa); argv es un apuntador hacia las diferentes cadenas de caracteres que componen la linea de comanda (nombre del programa y parámetros)

```
prog 1 23 taratata
```

En este caso argc=4, argv[0]=“prog”, argv[1]=“1”, argv[2]=“23”, argv[3]=“taratata”

Apuntadores sobre funciones

A veces puede ser útil poder usar dentro de una función varias funciones para hacer tal o tal tarea sin tener que reescribir el código correspondiendo a todos los casos.
Por ejemplo, suponemos que queremos, dentro de una función, hacer ordenamiento (sort) de números dentro de un arreglo; existen varios algoritmos : puede ser útil pasar como parámetro “la función” que lo hará.

Apuntadores sobre funciones

Para eso existe un apuntador sobre función que es simplemente un objeto apuntando hacia el código de la función en la memoria. Si la funcion es de prototipo :

```
tipo funcion(tipo1,...,tipon);
```

Entonces un apuntador hacia este tipo de funciones tiene este tipo :

```
tipo (*)(tipo1, ..., tipon);
```

Apuntadores sobre funciones

Ejemplo :

```
int applySort(int *, int, int(*)(int *, int));
```

Tomaría como entrada apuntador hacia datos enteros, numero de datos y una función de ordenamiento que regresa int y que toma como input, también, apuntador y numero de datos. Definiríamos :

```
int quickSort(int *data,int ndata) {
...
};
int heapSort(int *data,int ndata) {
...
};
```

Apuntadores sobre funciones

Ejemplo (seguida):
Luego lo podríamos usar así :

```
int *data;
int ndata;
...
int err=applySort(data,ndata,quickSort);
```

Numero variable de parámetros

- Es posible definir funciones que toman un numero variable de parámetros. Ya conocen unas : printf, scanf...
- En este caso, el prototipo debe de especificar al menos un parámetro formal, y los potenciales están resumidos por ... :

```
int varf(int a, char c, ...);
```

o también :

```
int printf(char *format, ...);
```

Numero variable de parámetros

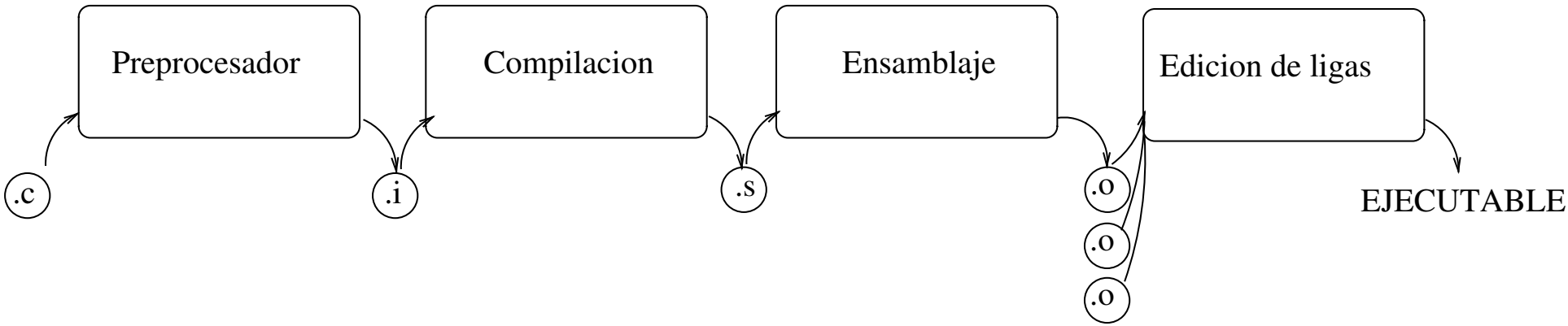
Para acceder a los parámetros, se usa macros definidas en stdarg.h

```
int sum(int ,...);

int sum(int num,...) {
  int res = 0;
  int i;
  va_list listaParams;
  va_start(listaParams, num);
  for (i = 0; i < num; i++)
    res += va_arg(listaParams, int);
  va_end(listaParams);
  return(res);
}
```

Outline

Preprocesador



El preprocesador transforma el código en otro código. Funciona por directivas que pueden :

- Incluir el contenido de otros archivos (#include).
- Definir constantes (#define).
- Definir macros (#define).
- Permitir compilación condicional (#if, #ifdef).

Las directivas al preprocesador se introducen por el símbolo "#".

Preprocesador

No hace fundamentalmente nada determinante en cuanto al código objeto generado, solo permite facilidades de escritura en el código. Por ejemplo :

- Evitar repetir unas constantes numéricas largas.
- Definir una macro que permita remplazar una función corta (se gana memoria y tiempo).

La directiva #include

- Permite incluir el contenido de código de otro archivo.
- Sirve en particular para incluir declaraciones de funciones incluidas en archivos *header*.

```
#include <math.h>

double x=0.144;
double y=pow(x,3.5);
```

La directiva #include

Ejemplo : el contenido de math.h

```
/*****************************************************************************
 *                                Math Functions                             *
 *****************************************************************************/
extern double  acos( double );
extern float  acosf( float );
extern double  asin( double );
extern float  asinf( float );
extern double  atan( double );
extern float  atanf( float );
extern double  atan2( double, double );
extern float  atan2f( float, float );
extern double  cos( double );
extern float  cosf( float );
extern double  sin( double );
extern float  sinf( float );
extern double  tan( double );
extern float  tanf( float );
extern double  acosh( double );
extern float  acoshf( float );
extern double  asinh( double );
extern float  asinhf( float );
```

La directiva #include

Dos sintaxis posibles :

1. Para la inclusión de archivos de la biblioteca estándar (math.h, stdio.h...) :

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

2. Para la inclusión de archivos que están localizados en el mismo directorio que el código compilado (o en los directorios especificados a la compilación por "-I")

```
#include "miPrograma.h"
```

La directiva #define

#define nombre cualquieracosa

En su primer uso, sirve para definir constantes : el preprocesador reemplaza cada ocurrencia de “nombre” por lo que sigue en la definición, “cualquieracosa” (que puede ser cualquiera cosa, incluso palabras llaves, elementos de sintaxis, constantes numericas. . .).

La directiva #define

Se pueden combinar :

```
#define NUM_LINEAS 480
#define NUM_COLUMNAS 640
#define TAMAÑO_IMAGENES NUM_LINEAS * NUM_COLUMNAS
```

La directiva #define

Hay un cierto número de constantes definidas ya por #define por el compilador :

Constante	**Uso**
__LINE__	Numero de la linea corriente
__FILE__	Nombre del archivo compilado
__DATE__	Fecha de la compilación %
__TIME__	Hora de la compilación %

La directiva #define

Ejemplo :

```
#include <stdio.h>

int main() {
   printf("Error en linea %d del archivo %s\n",
          __LINE__,__FILE__);
   return 0;
}
```

Da :

```
Macaye[CLASE5][21:23]>./test
Error en linea 5 del archivo test.c
```

La directiva #define

Cuidado : el preprocesador va a remplazar cada ocurrencia del texto, entonces, puede eventualmente llevar a problemotas :

```
#define num 10

void func1() {
        int numCosas;
        ...
}
```

(una variable no puede empezar por valores numéricos !)
Evitar pues que el nombre sea demasiado “frecuente”.

La directiva #define

Macros : puede considerar argumentos al nombre que se va a cambiar, y actuar como "función" (pero no tiene nada que ver con funciones C !)

```
#define max(x,y) (x>y?x:y)
#define min(x,y) (x<=y?x:y)
```

En general

```
#define nombre(lista-parametros) cuerpo-de-la-macro
```

Macros : cuidado

Las macros están consideradas como macros a partir del carácter de pareentesis que sigue inmediatamente el nombre de la macro : no dejar espacio.

```
#define cuadrado (x) (x * x)
```

Macros : cuidado

Efectos de bordo :

```
#define cuadrado(x) ((x) * (x))

int b = cuadrado(++a);
```

Macros : cuidado

Efectos de bordo :

```
#define cuadrado(x) (x * x)

int c = cuadrado(a+b);
```

Macros : cuidado

Evitar el abuso... (Algol-like, en Bourne shell)

```
#define IF      if(
#define THEN    ){
#define ELSE    } else {
#define ELIF    } else if (
#define FI      ;}
#define ENDSW   }
#define FOR     for(
#define WHILE   while(
#define LOOP    for(;;){
#define POOL    }
assign(n,v)
        NAMPTR          n;
        STRING          v;
{
        IF n->namflg&N_RDONLY
        THEN    failed(n->namid,wtfailed);
        ELSE    replace(&n->namval,v);
        FI
}
```

Compilación condicional

```
#if condicion-1
   pedazo-1
#elif condicion-2
   pedazo-2
      ...
#elif condicion-n
   pedazo-n
#else
   pedazo-else
#endif
```

Compilación condicional

Cuando se quiere hacer pruebas, o dejar un pedazo de código sin compilarlo :

```
#if 0
// Este pedazo no lo terminé entonces
// le dejo asi, no estará compilado
int x = blabla;
#endif
```

Se puede checar el valor de una constante definida por un #define

```
#if PROCESADOR == ALPHA
    tamano_long = 64;
#endif
```

Compilación condicional

Se puede checar la existencia de un símbolo definido con #define con #ifdef.

```
#define DEBUG
    ....
#ifdef DEBUG
          // Aquí harías cosas que servirían
          // para debogar
#endif /* DEBUG */
```

Compilación condicional

Existe también el operador #defined que va con un #if y que tiene la ventaja de puede ir por varias instancias a la vez :

```
#if defined(LINUX) || defined(DARWIN)
```

Compilación condicional : #include

A veces se compilan varios archivos a la vez; en este caso los #include pueden ser fuentes de error, por multiplicidad de las declaraciones de funciones en los *headers*. Para evitar eso, los archivos están marcados con #define (guardias) :

```
#ifndef _MYPROG_H_
#define _MYPROG_H_

...

#endif
```

Compilación condicional : #include

Ejemplo :

```
/*************************************************
 *                                               *
 *       File:   math.h                          *
 *                                               *
 *************************************************/
#ifndef __MATH__
#define __MATH__
```

La directiva #pragma

Es una directiva que se usa para dar opciones al compilador; las comandos que se pueden pasar a éste a través de #pragma dependen fuertemente de la arquitectura. Ejemplo :

```
#pragma once
```

igual a los guardias de inclusión.

Outline

Acceso a los archivos

El C permite abrir archivos, leerlos y escribirlos, a través de una serie de funciones dedicadas : fopen, fclose, fread... y de una estructura FILE que están declaradas/definidas en <stdio.h>.

Buffers

Leer/escribir datos en la memoria es mucho más rápido que hacerlo en un soporte físico como disco duro. Por eso para evitar congestión el acceso a los archivos se hace por medio de un *buffer*, que es una zona de la memoria viva dedicada a guardar esos datos, por un tiempo.

La estructura FILE

Para manipular un archivo, se necesita saber donde está esta memoria *buffer*, donde está la cabeza de lectura, cual es el modo de lectura (escritura/lectura). Todas esas informaciones están en la estructura FILE, cuyo apuntador es llamado flujo (file stream).

La función fopen

Permite inicializar el flujo de datos, a partir del nombre del archivo (archivo en el disco, puerto de interfase...). Toma como argumentos el nombre de archivo y el modo (ambos son cadenas de caracteres) y regresa el flujo de datos abierto si OK, NULL si no.

La función fopen

Se hace una distinción entre los archivos texto y archivos binario : en el segundo caso, no se toma en cuenta los caracteres especiales, como \n. Para indicar cual modo se considera, se usa el segundo argumento de fopen.

La función fopen : modos de acceso

"r"	archivo texto en lectura
"w"	archivo texto en escritura
"a"	archivo texto en escritura al final (*append*)
"rb"	archivo binario en lectura
"wb"	archivo binario en escritura
"ab"	archivo binario en escritura al final
"r+"	archivo texto en lectura/escritura
"w+"	archivo texto en lectura/escritura
"a+"	archivo texto en lectura/escritura al final
"r+b"	archivo binario en lectura/escritura
"w+b"	archivo binario en lectura/escritura
"a+b"	archivo binario en lectura/escritura al final

La función fopen

```
#include <stdio.h>
FILE *fp;
...
if ((fp = fopen("donnees.txt","r")) == NULL) {
    fprintf(stderr,"No puedo abrir el archivo\n");
    exit(1);
}
```

La función fopen : flujos estándar

A la ejecución de todo programa, tres flujos están abiertos sistemáticamente :

- stdout (salida estándar) : pantalla, por default,
- stdin (entrada estándar) : teclado, por default,
- stderr (salida de error estándar) : pantalla por default.

Se pueden redirigir hacia otros flujos (mecanismo de pipe).

La función fclose

Cierra un flujo abierto :

```
fclose(fp);
```

Regresa 0 si todo OK.

La función fprintf

Escritura formateada en un archivo :

```
float a,b;
int i;
fprintf(fp,"%f␣%f␣%d",a,b,i);
```

Ver la clase 2., la sintaxis es igual a printf.

La función fscanf

Lectura formateada desde un archivo :

```
float a,b;
int i;
fscanf(fp,"%f␣%f␣%d",&a,&b,&i);
```

Ver la clase 2., la sintaxis es igual a scanf.

Las funciones fprintf y fscanf

El comando printf es completamente equivalente a fprintf(stdout,), y scanf a fscanf(stdin,)

La función fflush

La función fflush fuerza a que se vacíe el contenido del buffer hacia el archivo meta.

```
fflush(fp);
```

Puede ser útil en situaciones en que queremos que los datos estén escritos en su destino final mientras se ejecuta el programa (y sin que nos importe que cueste tiempo esta escritura).

La función fflush

```
If the given stream was open for writing and the last i/o
operation was an output operation, any unwritten data
 in the output buffer is written to the file.
If the stream was open for reading, the behavior depends
 on the specific implementation. In some implementations this
 causes the input buffer to be cleared.
If the argument is a null pointer, all open files are flushed
The stream remains open after this call.
When a file is closed, either because of a call to fclose or
 because the program terminates, all the buffers associated
 with it are automatically flushed.
```

La función fflush

```
#include <stdio.h>

int main() {
  char x, y;
    printf( "First: " );
    scanf( "%c", &x );

    printf( "Second: " );
    scanf( "%c", &y );
    printf( "\n\n%c, %c ", x,y);
    return 0;
 }
```

La función fflush

```
#include <stdio.h>

int main() {
  char x, y, c;
   printf( "First: " );
   scanf( "%c", &x );

   while ((c = fgetc(stdin)) != '\n') {}

   printf( "Second: " );
   scanf( "%c", &y );
   printf( "\n\n%c, %c ", x,y);
   return 0;
}
```

La función setbuf

```
char buffer[BUFSIZ];
setbuf(fp1,buffer);
setbuf(fp2,NULL);
```

Para controlar la zona buffer sí mismo o indicar de no utilizar buffer (con NULL).

Las funciones sobre caracteres

```
// Get a character
int fgetc(FILE* flujo);
 // Put a character
int fputc(int caracter, FILE *flujo);
// Push back a character in the stream
int ungetc(int caracter, FILE *flujo);
```

Las funciones sobre caracteres

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
   FILE *fin;
   int c;
   if ((fin = fopen("data.txt","rb")) == NULL) {
        fprintf(stderr, "\nError al abrir el archivo");
        return(1);
     }
   while ((c = fgetc(fin)) != EOF)
     fputc(c,stdout);
   fclose(f_in);
   return(0);
}
```

Las funciones sobre caracteres

```
#include <stdio.h>
int main () {
  FILE * pFile;
  int c;
  char buffer [256];
  pFile = fopen ("myfile.txt","rb");
  if (pFile==NULL) perror ("Error opening file");
  else {
    while (!feof (pFile)) {
      c=getc (pFile);
      if (c == '#') ungetc ('@',pFile);
      else ungetc (c,pFile);
      fgets (buffer,255,pFile);
      fputs (buffer,stdout);
    }
  }
```

Escritura y lectura binarias

Para ir a leer/escribir directamente los datos sin pasar por representaciones.

```
size_t fread(void *ptr,
             size_t tam, size_t num, FILE *flujo);
size_t fwrite(void *ptr,
             size_t tam, size_t num, FILE *flujo);
```

Toman como argumentos el número de octetos de la estructura que leer y el numero de estructuras que leer.

Escritura y lectura binarias

Lectura del *header* de un archivo MIDI.

```
char firstchain[4];
// read first 4 bytes
fread(&firstchain[0],1,4,f);
if (firstchain[0]!='M' ||
    firstchain[1]!='T' ||
    firstchain[2]!='h' ||
    firstchain[3]!='d') {
  fprintf(stderr,"This is not a MIDI header");
  return 1;
}
```

Escritura y lectura no secuenciales

```
int fseek(FILE *flujo, long desplac, int ref);
```

Para moverse de desplac octetos a partir de la referencia ref. Sus valores son SEEK_SET (principio del archivo), SEEK_CUR (posición actual) y SEEK_END (final del archivo)

```
void rewind(FILE * flujo);
```

Para reposicionarse al principio, casi equivalente a

```
fseek(flujo,0L,SEEK_SET);
```

Escritura y lectura no secuenciales

```
long int ftell( FILE *flujo);
```

Valor de la posición: numero de bytes en el caso de manipulación binaria, nada garantizado en caso de manipulación de texto.

Escritura y lectura no secuenciales

```
/* fread example: read a complete file */
#include <stdio.h>
#include <stdlib.h>
int main () {
  FILE * pFile;
  long lSize;
  char * buffer;
  size_t result;
  pFile = fopen ( "myfile.bin" , "rb" );
  if (pFile==NULL) {fputs ("File error",stderr); exit
  // obtain file size:
  fseek (pFile , 0 , SEEK_END);
  lSize = ftell (pFile);
  rewind (pFile);
```

Escritura y lectura no secuenciales

```
  // allocate memory to contain the whole file:
  buffer = (char*) malloc (sizeof(char)*lSize);
  if (buffer == NULL) {fputs ("Memory error",stderr);

  // copy the file into the buffer:
  result = fread (buffer,1,lSize,pFile);
  if (result != lSize) {fputs ("Reading error",stderr

  /* the whole file is now loaded in the memory buffe

  // terminate
  fclose (pFile);
  free (buffer);
  return 0;
}
```