Programación : repaso de C (5)

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Agosto 2009

Outline

Outline

Caracteres

- En general, sobre un octeto.
- Un carácter ASCII : sobre 7 bits.
- Un carácter ISO-8859 : sobre 8 bits (ISO-8859-1 : caracteres de las lenguas de Europa occidental).
  Ejemplo : Ú $\leftrightarrow$ 233
- Unicode : otro estándar, con representaciones en series de 8, 16 y 32 bits (UTF-8,UTF-16,UTF-32). En C, necesita usar tipo especial de caracteres (wchar_t) y funciones especiales de I/O.

Lectura de cadenas de caracteres

Desde flujos de datos :

```
char *fgets(char *str, int size, FILE *stream);
char *gets(char *str);
```

The fgets() function reads at most one less than the number of characters specified by size from the given stream and stores them in the string str. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. If any characters are read and there is no error, a '\0' character is appended to end the string.

The gets() function is equivalent to fgets() with an infinite size and a stream of stdin, except that the newline character (if any) is not stored in the string. It is the caller's responsibility to ensure that the input line, if any, is sufficiently short to fit in the string.

Lectura de cadenas de caracteres

```
#include <stdio.h>
int main() {
   char test[10];
   fgets(test,10,stdin);
   fprintf(stderr,"%s\n",test);
}
```

Da :

```
Macaye[CLASE5][20:21]>./test2
Eso es una prueba
Eso es un
```

Escritura/lectura en cadenas de caracteres

```
int sprintf(char *cadena, const char *format, ...);
int sscanf(char *cadena, const char *format, ...);
```

Sintaxis igual a fprintf. Requieren, como todas las funciones sobre cadenas de caracteres, la inclusión de string.h.

Comparación de cadenas de caracteres

```
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n
```

Compara cadenas lexicográficamente, y regresa > 0 si $s1 > s2$, igual a 0 si son iguales etc... Sobre las cadenas enteras o sobre los n primeros caracteres.

Copia y concatenación

```
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t len);
```

Añade la cadena *s*2 a la cadena *s*1 y regresa *s*1.

```
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t len);
```

Copia la cadena s2 (incluso el "\ 0") en s1. Cuidado que con strncpy en el caso que s2 es mas largo que s1, la cadena s2 no tendría el \ 0 final.

Otros

```
char *strchr(const char *s, int c);
```

Localiza el carácter *c* en la cadena y regresa apuntador hacia donde está en la cadena.

```
size_t    strlen(const char *);
```

Tamaño de la cadena (antes del carácter de terminación).

Outline

Programación en practica

Es muy fácil hacer con C código que no se pueda portar, que no se pueda reutilizar, que no se pueda entender. Contra eso :

- Organizar su código.
- Usar funciones.
- Usar nombres adecuados.
- Comentar.

Separación del código

- Separar el código en varias porciones, cada una relacionada con un tema, una estructura particular (en unidades conceptuales, "objetos").
- Dedicar archivos .c, eventualmente, a cada porción.
- Poner los prototipos, declaración de constantes en archivos *header* .h

Separación del código

Separar las funcionalidades y estructuras de una instancia de programa (el main) en que se usan : usar bibliotecas ! (bibliotecas estáticas/bibliotecas dinámicas)

Separación del código

Ejemplo :

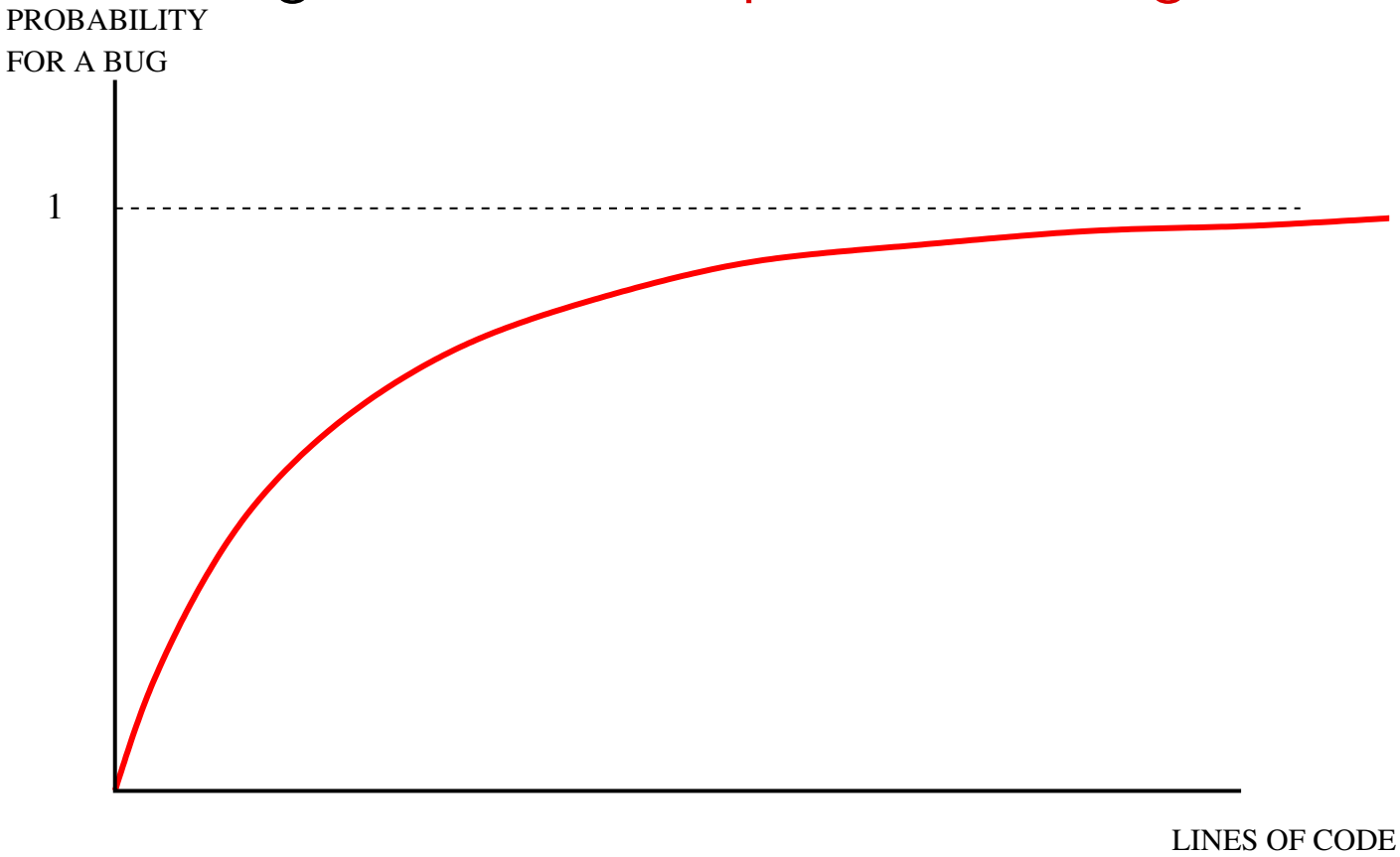
```
image.h
image.c
imageDerivative.h
imageDerivative.c
interestPoints.h
interestPoints.c
------------------------
imageLib.dll (.so,.a,.dylib)

prog.c con un main dedicado a un uso en particular
```

Usar ANSI-C

En la medida de lo posible, no usar cosas especificas de su sistema, de su compilador. . . Hay que pensar que su código puede estar probado por gente que corren otro OS que el suyo, o otra versión. . .

Minimizar la cantidad de código !

- Cada linea de código es una fuente potencial de bug.



- Evitar repeticiones de código !

Minimizar la cantidad de código !

Usar librerías ya hechas y blindadas : se salva tiempo y estarán probablemente mejor optimizadas.

- Lapack, escrita en Fortran, sigue una referencia para computo numérico
- Boost intenta proponer librerías C++ de estructuras, contenedores...
- OpenCV, libMagick en procesamiento de imágenes y visión artificial.

Legibilidad

- Elegir nombres de variables que “hablan” : si son cantidades físicas, por ejemplo, debería de verse en el nombre.
- Tampoco usar nombres demasiado largos.
- Elegir una convención tipográfica y conservarla. Tradicionalmente, caracteres mayúsculos para constantes definidas por define, minúsculas para variables.

Guacala

Tomado en la pagina del ioccc

```
#include <ncurses.h>/****************************************************/
              int                  m[256                    ]  [          256    ],a
 ,b    ;;;    ;;;    WINDOW*w;    char*l=""    "\176qxl"    "q"    "q"    "k"    "w\
xm"    "x"    "t"           "j"          "v"          "u"          "n"          ,Q[
 ]=    "Z"    "pt!ftd`"     "qdc!`eu"    "dq!$c!nnwf"/**    ***    */"t\040\t";c(
int    u  ,         int          v){                          v?m    [u]          [v-
 1]    |=2,m[u][v-1] &   48?W][v-1    ] &    15]]):0:0;u?m[u    -1][v]|=1    ,m[
 u-                 1][    v]&          48?                  W-1    ][v          ]&
15]    ]):0:0;v<    255    ?m[    u][v+1]|=8,m[u][v+1]&    48?    W][    v+1]&15]]
):0          :0;           u <                255     ?m[    u+1              ][v    ]|=
4,m[u+1][    v]&48?W+1][v]&15]]):0:0;W][    v]&    15]    ]);}cu(char*q){    return
 *q                 ?cu    (q+           1)&           1?q    [0]                ++:
q[0    ]--    :1;    }d(    int    u  ,    int/**/v,    int/**/x,    int    y){    int
Y=y    -v,    X=x           -u;    int           S,s    ;Y<           0?Y    =-Y    ,s,
s=-    1:(    s=1);X<0?X=-X,S    =-1    :(S=    1);    Y<<=    1;X<<=1;    if(X>Y){
int    f=Y                  -(X    >>1    );;                  while(u!=              x){
f>=    0?v+=s,f-=X:0;u    +=S    ;f+=    Y;m[u][v]|=32;mvwaddch(w,v    ,u,    m[u
 ][               v]&    64?    60:           46)           ;if    (m[    u][
v]&16){c(u,v);;    ;;;    ;;;    return;}}    }else{int    f=X    -(Y>>1);;    while
 (v    !=y         ){f    >=0           ?u    +=S,                  f-=           Y:0
 ;v    +=s    ;f+=X;m[u][v]|=    32;mvwaddch(w,v    ,u,m[u][v]&64?60:46);if(m[u
 ][                 v]&           16)    {c(    u,v                              );
 ;    return;;;}}}}Z(    int/**/a,    int    b){    }e(    int/**/y,int/**/    x){
int                 i  ;          for           (i=                a;i              <=a
+S;i++)d(y,x,i,b),d(y,x,i,b+L);for(i=b;i<=b+L;i++)d(y,x,a,i),d(y,x,a+    S,i
 );                        ;;;           ;;;           ;;;           ;;;    ;
  mvwaddch(w,x,y,64);    ;;;    ;;;    ;;;    prefresh(    w,b,a,0,0    ,L-    1,S-1
);}               main(           int                  V ,    char                *C[
 ]    ){FILE*f=    fopen(V==1?"arachnid.c"/**/    :C[    1],"r");int/**/x,y,c,
```

Legibilidad

No usar apuntadores cuando no es necesario, y aunque todo sea sintácticamente correcto.

- Complica su vida y puede ser fuerte fuente de error.
- Complica la lectura para los otros.

Legibilidad

Evitar la redefinición de constantes numéricas : definirlas una vez por todas, visibles por todo el código.

- En C, usar los #define (con mayúsculas). Un ejemplo de libdc1394

```
/* Capture flags */
#define DC1394_CAPTURE_FLAGS_CHANNEL_ALLOC    0x00000001U
#define DC1394_CAPTURE_FLAGS_BANDWIDTH_ALLOC  0x00000002U
/* a reasonable default value: do alloc of bandwidth and channel
#define DC1394_CAPTURE_FLAGS_DEFAULT          0x00000004U
```

- En C++, los static const

En practica sería bien no tener un solo valor numérico dentro del código si mismo.

Legibilidad

Aunque los enum no estén verificados bien, sirven para aumentar la legibilidad, cuando un parámetro toma un numero conocido de valores definidas correspondiendo a una semántica :

```
/* Operation modes */
typedef enum {
  DC1394_OPERATION_MODE_LEGACY = 480,
  DC1394_OPERATION_MODE_1394B
} dc1394operation_mode_t;

/* Format 7 sensor layouts*/
typedef enum {
  DC1394_COLOR_FILTER_RGGB = 512,
  DC1394_COLOR_FILTER_GBRG,
  DC1394_COLOR_FILTER_GRBG,
  DC1394_COLOR_FILTER_BGGR
} dc1394color_filter_t;
```

Legibilidad

- Una instrucción por linea.
- Indentación : elegir una regla y respetarla (en general los editores vienen con indentación automática de código).

Comentarios

- Hay que pensar que su código estará leído por varia gente que van a querer entender de que se trata en tu programa !
- Comentar las partes críticas, que puedan ser difíciles que leer.
- Usar también comentarios para describir que hace, generalmente, el programa, la librería (o la clase C++), en introducción; eventualmente la licencia bajo cual viene tu programa.

Comentarios

Usar comentarios en los headers para describir, antes de cada definición de función, lo que hace esta función.

```
/* Retrieves next contour */
CVAPI(CvSeq*)  cvFindNextContour(CvContourScanner scanner );


/* Substitutes the last retrieved contour with the new one
   (if the substitutor is null, the last retrieved contour is
    removed from the tree) */
CVAPI(void)    cvSubstituteContour(CvContourScanner scanner,
                                   CvSeq* new_contour );


/* Releases contour scanner and returns pointer to the first
outer contour */
CVAPI(CvSeq*)  cvEndFindContours(CvContourScanner* scanner );
```

Comentarios

En las partes con estructuras de control, eventualmente, se puede comentar los { para recordar a que control pertenece :

```
for( i = 0; i < _cascade->count; i++ )
  {
      int j, k, l;
      for( j = 0; j < cascade->stage_classifier[i].count; j++ )
      {
          for( l = 0; l < cascade->stage_classifier[i].classifier[j
              CvHaarFeature* feature =
                  &_cascade->stage_classifier[i].classifier[j].haa
              [...]
              hidfeature->rect[0].weight = (float)(-sum0/area0);
          } /* l */
      } /* j */
  } /*i*/
```

Outline

Generación automatica de documentación

Un software como doxygen usa los comentarios que encuentra en los *headers* de tu código para generar automáticamente documentación HTML, PDF, ... con indices de palabras. En C++, hasta sale automáticamente la estructura de clases de tu programa, gráficas de relaciones...
`http://www.doxygen.org`

Debugger

Importante (pero no necesariamente suficiente) para encontrar bugs : se ejecuta el programa a través del debugger, que tiene control sobre la memoria.

- Ejecución paso por paso, linealmente o entrando en funciones.
- Puntos de parada (*breakpoints*).
- Observación de las variables, de una zona memoria, de registros de la CPU, de la pila corriente (con llamadas a funciones)...
- Integrado en la mayoría de los entornos de programación (Visual Studio, Kdevelop...) : GDB, ...

Verificador de memoria

Son herramientas que van aun mas adelante de lo que hacen los debugger clásicos, ya que verifican todas las alocaciones/liberaciones de memoria, la no-inicializacion de variables. . . y detecta posibles fugas de memoria (*memory leaks*) : purify (IBM, sobre casi todas plataformas), valgrind (libre, en Linux con proyectos de portarlo en MacOSX)

Frente a un big bug...

- Verificar todas las alocaciones/liberaciones del programa.
- Usar el debugger para examinar el stack al momento del bug.
- Compilar el programa con los simbolos (gcc -g).
- Evitar caer en Panic Mode.

Manejar errores

Propagar errores, por códigos por ejemplo, usando el valor de regreso de las funciones

```
/* Return values for visible functions*/
typedef enum {
  DC1394_SUCCESS = 0,    /* Success is zero */
  DC1394_FAILURE,        /* Errors are positive numbers */
  DC1394_NO_FRAME = -2, /* Warnings or info are negative numbers */
  DC1394_NO_CAMERA = 3,
  DC1394_NOT_A_CAMERA,
  DC1394_FUNCTION_NOT_SUPPORTED,
  DC1394_CAMERA_NOT_INITIALIZED,
  DC1394_INVALID_FEATURE,
  DC1394_INVALID_VIDEO_FORMAT,
  DC1394_INVALID_VIDEO_MODE,
  DC1394_INVALID_FRAMERATE,
  DC1394_INVALID_TRIGGER_MODE,
  DC1394_INVALID_TRIGGER_SOURCE,
  DC1394_INVALID_ISO_SPEED,
  DC1394_INVALID_IIDC_VERSION,
  DC1394_INVALID_COLOR_CODING
  ...
```

Manejar errores

En cualquier caso, es muy importante verificar el valor de regreso de toda función, en particular de las de alocacion de memoria.

```
int *data = (int *)malloc(1000*sizeof(int));
if (data==NULL) {
        ...
```

Excepciones

No existen en C pero se puede re-escribirlas o usar software ya existente para manejarlas ; vienen por default en C++.

Son interrupciones dirigidas de la ejecución del programa que re-orientan la ejecución a otras partes que están dedicadas a manejar los diferentes casos. Son útiles porque permiten separar claramente el código del funcionamiento normal del código dedicado al procesamiento de las errores. Las veremos mas en detalles en las clases de C++.

Sistemas de versioning

Para “grandes” proyectos (una biblioteca que desarrollas durante tu maestría o tu doctorado), es muy útil versionar tu código, o sea usar herramientas que guardan (por diferencias) todas las versiones de tu código :

- Puedes regresar fácilmente a versiones previas (por ejemplo si te das cuenta que todo el código que escribiste desde una semana es tonto).
- Puedes ver la evolución del código, las diferencias entre las versiones, entre fechas y así detectar mas fácilmente un *bug* potencial.
- Puedes colaborar con alguien mas...

Usar sistemas de versioning

Dos herramientas dominan :

- CVS, la mas antigua
  `http://www.nongnu.org/cvs/`
- SVN (subversion), reciente, y de funcionamiento muy similar a CVS, con desarrollo pendiente
  `http://subversion.tigris.org/`
- git, también reciente
  `http://git-scm.com/`

Outline

Errores simples

- Confundir "=" y "==".
- Olvidar un `break` en un `switch`.
- Indices en los arreglos (entre 0 y $n-1$ incluido).
- Pensar que todo es inicializado a 0.

Errores simples

- División entera.

```
double ratio = 1/3;
```

- Ciclos infinitos.

```
int i = 10;
while( i > 0 );
 i--;
```

- Prototipos : cuando encuentra una función con prototipo no definido, por default considera que regresa int; la compilación puede funcionar (con Warnings) pero habrá problemas.

```
double a = sqrt(2);
```

Errores simples

- Con if... else... : mejor usar llaves.

```
if (cosa)
        if (otraCosa)
                hazCosa();
else
        hazOtraCosa();
```

- Variables globales : si su nombre es demasiado “común”, pueden estar confundidas con variables locales de mismo nombre.
- Orden de las operaciones :

```
data1[j++]=data2[j++]
```

Errores con apuntadores

- Apuntadores no inicializados.
- Indices fuera del rango admisible (cuando los indices vienen de un proceso de calculo, prever una función para verificarles.
- Apuntadores leídos con otro tipo con el que han sido creados.

Errores sobre fscanf

Las funciones de tipo scanf son una gran fuente de error, por dos razones :

- porque toman como ultimo argumento un apuntador hacia la estructura que almacenará la cosa leida,
- porque hay que usar el buen operando de formataje (%*lf* para dobles),
- peligro con el formato %*s* : la cadena pasada puede estar insuficiente (problemas de memoria).

Errores con cadenas de caracteres

- Confundir el 'a' con "a".
- Usar el "==" o cualquier operador de relación para comparar dos cadenas de caracteres.
- Olvidar el carácter de terminación "\0" o no prever espacio para este en una alocación memoria para cadenas de caracteres.

Cuando ese ultimo tipo de error ocurre, puede ser que no pasa nada al momento pero que la función siguiente falla !

Errores con archivos

Test con EOF :

```
int count_line_size( FILE * fp ) {
  char ch;
  int  cnt = 0;

  while( (ch = fgetc(fp)) != EOF && ch != '\n')
    cnt++;
  return cnt;
}
```

Errores con archivos

Cuidado al valor de regreso de fgetc, getc, getchar !

```
NAME
       fgetc, getc, getc_unlocked, getchar, getchar_unlocked, getw -- g
       character or word from input stream

LIBRARY
       Standard C Library (libc, -lc)

SYNOPSIS
      #include <stdio.h>
      int
      fgetc(FILE *stream);
      int
      getc(FILE *stream);
      int
      getchar();
```

Errores con archivos

Mal uso de feof

```
#include <stdio.h>
int main() {
   FILE * fp = fopen("test.txt", "r");
   char line[100];
   while( ! feof(fp) ) {
      fgets(line, sizeof(line), fp);
      fputs(line, stdout);
   }
   fclose(fp);
   return 0;
}
```

feof da el resultado de la ultima operación de lectura (fgets). Mejor verificar directamente el resultado de fgets.

Errores con archivos

Buffer no vacío.

```
int x;
char st[30];

printf("Enter an integer: ");
scanf("%d", &x);
printf("Enter a line of text: ");
fgets(st, 30, stdin);
```

Errores con archivos

Buffer no vacío : la función fgets va a empezar por leer un carácter "$\backslash n$" dejado por el previo scanf. Cuidado a los scanf múltiples : prever una función para terminar de leer todos los caracteres de un stream abierto en lectura.

Outline

Antes de optimizar

Donald Knuth:
"Premature optimization is the root of all evil (or at least most of it) in programming"
Además, demasiado optimización hace el código poco leíble.

Analizar los algoritmos

Es la primera cosa que hacer : la tendencia general del tiempo de ejecución depende de unas variables (de orden de magnitud n, por ejemplo) que hay que determinar (tamaño de un arreglo de datos) y es una función de esas variables. Las elecciones que ustedes hacen para diseñar sus algoritmos van a influir sobre esta función : se suele examinar de que rango está con respeto a n; $O(\log n)$; $O(n)$, $O(n^2)$, $O(e^n)$

No sirve mucho ir a optimizar "por el código" antes de haber optimizado conceptualmente su algoritmo !

Profiling

Existen unas herramientas llamadas *profilers* que te dan cuentas de cuanto tiempo tomo cada función... : son útiles para saber qué parte de su programa optimizar en prioridad (*bottlenecks*).
Ejemplo : gprof (Linux).

Funciones

- Cuidar todas las funciones : no debe de haber argumentos pesados pasados por valor; si es el caso, hacerles apuntadores !
- Minimizar la cantidad de variables locales.
- Usar void si no valor de regreso requerido.

Variables

- Para multiplicación y división de enteros por potencias de 2, es la manera mas rápida. Ahora, hay que cuidar los efectos de bordo (en particular el signo).
- Si posible, poner todos tipos enteros en int (¿por qué ?).
- Arreglos de estructuras : usar tamaños múltiplos de potencias de 2 (¿por qué ?)

Palabra llave register

Este modificador hace que la variable, si es posible, está guardada en un registro de la CPU, y no en memoria. Es útil si, localmente, esta variable está usada muchas veces (por ejemplo, un indice de un arreglo en ciclos). Es posible que compiladores eficientes pongan esta palabra llave por si mismos. Por definición, no se puede entonces usar el operador &.

Palabra llave inline

Es reciente en C (C99) y esta tomada del C++; toma (al nivel del ensamblador) el código de la función y lo pega donde la función esta llamada, haciendo optimización entre los dos cuerpos de las funciones, la llamada y la que llama.

```
inline double sum(const double * numbers, const int
        double s = 0;
        int n;
        for(i = 0; i <n; i++)
                s += numbers[i];
        return s;
}
```

Usar para funciones : cortas ; llamadas un numero importante de veces.

Casos mas frecuentes primero !

En los procesamientos de switch, en las condiciones, poner los casos mas frecuentes primero,

```
switch (cosa) {
        case cosaFrecuente:
                ...
                break;
        case cosaMenosFrecuente:
                ...
                break;
        ...
}
}
```

Optimización por el compilador

La mayoría de los compiladores pueden hacer un gran trabajo de optimización por sí mismo : investigar en la documentación de su compilador favorito !

Paralelización : threads

- Ahora muchas maquinas vienen con dobles/cuatruples corazones ; un programa lineal no usa estos dos corazones.
- Paralelizar estrictamente requiere crear procesos distintos, y hacerlos cambiar sus datos.
- Hay maneras menos complejas de hacer : los procesos ligeros, o *threads*, que tienen su propia pila sistema pero que comparten la memoria virtual con sus hermanos.
- Típicamente : dejar la interface gráfica a un *thread* y los cálculos a otros *threads*.

Paralelización : threads

+ Hay efectivamente paralelización.
+ Es fácil compartir información entre *threads*.
- Se necesita mecanismos para no escribir al mismo tiempo y mecanismos de sincronización.
- No hay estándar (mientras si en Java por ejemplo).

Paralelización : threads POSIX

Es una implementación de los *threads* desarrollada para sistemas Unix pero que también existe en Windows; la biblioteca se llama pthread.

Paralelización : threads POSIX

```
dataStruct datathreads[MAXTHREADS_MAX];
pthread_t threads[MAXTHREADS_MAX];
...
for(i = 0; i < nThreads; i++) {
   ...
   if(pthread_create(&threads[i], NULL,
                     functionThread,
                     (void*)&datathreads[i]) < 0) {
      fprintf(stderr, "Error creating pthread\n");
      exit(1);
   }
}
for(i = 0; i < nThreads; i++) {
   pthread_join(threads[i], &valBack);
}
```