Cours de C - IR1
2007-2008

Tableaux, fonctions et passages d'adresses

Sébastien Paumier

Les tableaux

- éléments contigus de même type
- déclaration: `type nom[taille];`

```
int t[N];
double cosinus[360];
```

- avec initialisation:

```
char vowel[6]={'a','e','i','o','u','y'};
```

- la taille devient optionnelle:

```
char* name[]={"Smith","Doe","X"};
```

Les tableaux

- type des éléments quelconque
- numérotation de `0` à `taille-1` ⚡

```
void init(int value[]) {
    int i;
    for (i=0;i<N;i++) {
        value[i]=i;
    }
}
```

Taille

- un tableau ne connaît pas sa taille
- l'opérateur **sizeof** ne marche que pour les tableaux dont la taille est connue à la compilation

```
void test_sizeof(int t[]) {
printf("sizeof(t)=%d bytes\n",sizeof(t));
}

int main(int argc,char* argv[]) {
int m[15];
test_sizeof(m);
printf("m: %d bytes\n",sizeof(m));
return 0;
}
```

→

```
$>./a.out
sizeof(t)=4 bytes
m: 60 bytes
```

Taille

- 3 solutions:
  - connaître la taille grâce à une constante
  - passer la taille en paramètre
  - utiliser un élément marqueur comme le `'\0'` pour les chaînes

```
void print1(int t[]) {
int i;
for (i=0;i<N;i++) {
   printf("%d\n",t[i]);
}
}
```

```
void print2(int t[],
            int n) {
int i;
for (i=0;i<n;i++) {
   printf("%d\n",t[i]);
}
}
```

```
void print3(int t[]) {
int i;
for (i=0;t[i]!=Z;i++) {
   printf("%d\n",t[i]);
}
}
```

Débordement

- aucun contrôle de débordement
- attention aux surprises!

```c
#define N 10

void foo(int A[],int B[]) {
   int i;
   for (i=0;i<20;i++) {
      B[i]=33;
   }
   for (i=0;i<N;i++) {
      printf("%d %d\n",A[i],B[i]);
   }
}
```

```
$>./a.out
33 33
33 33
33 33
33 33
33 33
33 33
33 33
33 33
2293600 33
2009000225 33
```

Tableaux à *n* dimensions

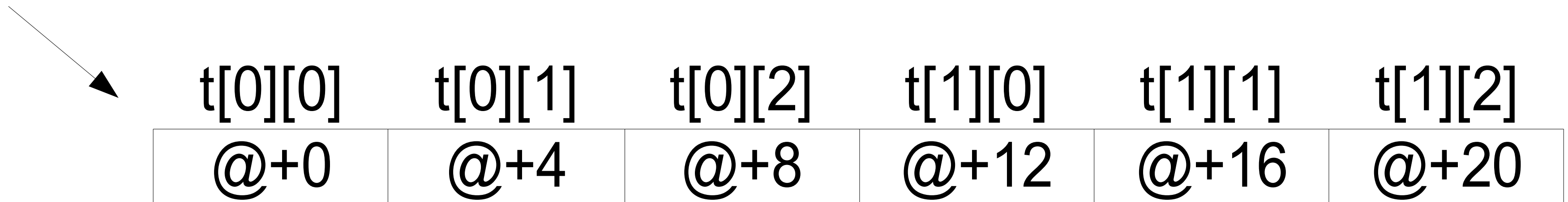
- `int t[100][16][45];`
- chaque `t[i][j]` est un tableau de 45 int
- quand on passe un tableau à une fonction, on doit mettre toutes les dimensions sauf la première :

```
void foo(int t[][M]) {
    /*
      ...
    */
}
```

```
void foo(int t[][]) {
    /*
      ...
    */
}
```

Tableaux à *n* dimensions

- `int t[2][3];`



t[0][0]	t[0][1]	t[0][2]	t[1][0]	t[1][1]	t[1][2]
@+0	@+4	@+8	@+12	@+16	@+20

- attention à l'ordre de parcours
  - pareil en théorie, pas en pratique:

```
for (i=0;i<2;i++)
  for (j=0;j<3;j++)
     ...t[i][j]...
```

```
for (j=0;j<3;j++)
   for (i=0;i<2;i++)
      ...t[i][j]...
```

Tableaux à *n* dimensions

```c
#define N 15000
char t[N][N];


int main(int argc,char* argv[]) {
   int i,j;
   time_t before=time(0);
   for (i=0;i<N;i++) {
      for (j=0;j<N;j++) {
         t[i][j]=1;
      }
   }
   time_t middle=time(0);
   printf("%d seconds\n",middle-before);
   for (j=0;j<N;j++) {
      for (i=0;i<N;i++) {
         t[i][j]=1;
      }
   }
   time_t end=time(0);
   printf("%d seconds\n",end-middle);
   return 0;
}
```

```
$>./a.out
4 seconds
32 seconds
```

Les fonctions

- une fonction a un prototype:
  - un nom
  - des paramètres
  - un type de retour
- exemple de définition:

```
float average(int t[]) {
   float sum=0;
   int i;
   for (i=0;i<N;i++) {
      sum=sum+t[i];
   }
   return sum/N;
}
```

void

- type spécial:
  - en valeur de retour: pas de valeur de retour

```
void print_sum(int a,int b) {
   /* ... */
}
```

  - en paramètre: pas de paramètres (facultatif)

```
char get_upper_char(void) {
   /* ... */
}
```

```
char get_upper_char() {
   /* ... */
}
```

Définition vs déclaration

- définition = code de la fonction
- déclaration = juste son prototype avec ;
- dans les .h
- dans un .c pour une fonction qu'on ne veut pas exporter

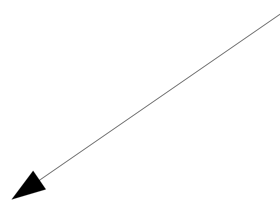
```c
void get_hen(void);

void get_egg(void) {
   /* ... */
   get_hen();
   /* ... */
}

void get_hen(void) {
   /* ... */
   get_egg();
   /* ... */
}
```

return

- quitte la fonction
- renvoie une valeur si retour ≠ `void`
- inutile seulement en fin de fonction `void`:

```
void print_sum(float a,float b) {
   printf("%f+%f=%f\n",a,b,a+b);
}
```

```
int print_sum2(float a,float b) {
   printf("%f+%f=%f\n",a,b,a+b);
}
```



```
$>gcc -Wall -ansi function.c
function.c: In function `print_sum2':
function.c:25: warning: control reaches end of non-void function
```

Valeurs de retour

- on peut ignorer une valeur de retour:
  - **printf**, **scanf**
- on ne peut pas utiliser une fonction **void** dans une expression:

```
void print_sum(float a,float b) {
   printf("%f+%f=%f\n",a,b,a+b);
}

int main(int argc,char* argv[]) {
   float s=print_sum(3.5,1.1);
   printf("sum=%f\n",s);
   return 0;
}
```

```
$>gcc -Wall -ansi function.c
function.c: In function `main':
function.c:8: void value not ignored as it ought to be
```

Le cas de main

- fonction particulière:
  - `return` quitte le programme et
  - renvoie le code de retour du programme
    - par convention: `0`=OK `≠0`=erreur
- si `main` est appelée explicitement par une fonction, `return` marche normalement

Paramètres de main

- **`int main(int argc,char* argv[]) {...}`**

Nombre de paramètres, y compris l'exécutable

Tableau de chaînes contenant les paramètres

```
int main(int argc,char* argv[]) {
int i;
for (i=0;i<argc;i++) {
   printf("arg #%d=%s\n",i,argv[i]);
}
return 0;
}
```

```
$>./a.out AA e "10 2"
arg #0=./a.out
arg #1=AA
arg #2=e
arg #3=10 2
```

Écrire une fonction

- réfléchir à l'utilité de la fonction
- 1 fonction = 1 seule tâche
- on ne mélange pas calcul et affichage!

```c
int minimum(int a,int b) {
    int min=(a<b)?a:b;
    printf("minimum=%d\n",min);
    return min;
}
```

```c
int minimum(int a,int b) {
    return (a<b)?a:b;
}

int main(int argc,char* argv[]) {
    int min=minimum(4,5);
    printf("min=%d\n",min);
    return 0;
}
```

Les paramètres

- ne pas mettre trop de paramètres!

```
int get_choice(char a,char c1,
               char c2,char c3,
               char c4,char c5) {
   if (a==c1 || a==c2 || a==c3 ||
       a==c4 || a==c5) return a;
   return -1;
}
```

```
int get_choice2(char a,char c[]) {
   int i;
   for (i=0;i<N;i++) {
      if (a==c[i]) return a;
   }
   return -1;
}
```

Définir le prototype

- de quoi la fonction a-t-elle besoin ?
- retourne-t-elle quelque chose ?
- y a-t-il des cas d'erreurs ?
- si oui, 3 solutions:
  - mettre un commentaire
  - renvoyer un code d'erreur
  - afficher un message et quitter le programme

Le commentaire

- utile quand la fonction n'est pas censée être appelée dans certains cas:

```c
/**
 * Copies the array 'src' into the 'dst' one.
 * 'dst' is supposed to be large enough.
 */
void copy(int src[],int dst[]);
```

```c
/**
 * Returns 1 if 'w' is an English word;
 * 0 otherwise. 'w' is not supposed to be
 * NULL;
 */
int is_english_word(char* w);
```

Le code d'erreur

- pas de problème si la fonction ne devait rien renvoyer:

```
int init(int t[],int size) {
    if (size<=0) return 0;
    int i;
    for (i=0;i<size;i++) t[i]=0;
    return 1;
}
```

- sinon, attention à la valeur choisie
  - on doit toujours pouvoir distinguer un cas d'erreur d'un cas normal

Le code d'erreur

- attention à la valeur:

```c
int minimum(int t[],int size) {
    if (size<=0) return -1;
    int min=t[0];
    int i;
    for (i=1;i<size;i++) {
        if (min<t[i]) min=t[i];
    }
    return min;
}
```

```c
/**
 * Returns the length of the given
 * string or -1 if NULL.
 */
int length(char* s) {
    if (s==NULL) return -1;
    int i;
    for (i=0;s[i]!='\0';i++);
    return i;
}
```

-1 : on ne peut pas savoir si on a une erreur ou si le minimum est -1

Le code d'erreur

- si toutes les valeurs possibles sont prises, il faut utiliser un passage par adresse pour le résultat ou pour le code d'erreur

```
int quotient(int a,int b,int *res) {
   if (b==0) return 0;
   *res=a/b;
   return 1;
}
```

L'interruption du programme

- <u>à n'utiliser que dans des cas très graves</u> ⚡
  - plus de mémoire
  - erreurs d'entrées-sorties
  - mauvais paramètres passés au programme
- message sur **stderr** + **exit(≠0);**

```
List* new_List(int n,List* next) {
   List* l=(List*)malloc(sizeof(List));
   if (l==NULL) {
      fprintf(stderr,"Not enough memory !\n");
      exit(1);
   }                /* On met un message parlant */
   l->n=n;          /* et non pas quelque chose  */
   l->next=next;    /* d'inutile comme "Error"   */
   return l;
}
```

Tests d'erreur

- quitter dès que possible en évitant les **else** inutiles
  - ⇨ améliore la lisibilité

```
int get_value(char* s) {
   int ret;
   if (s==NULL) {
      ret=-1;
   } else if (!isdigit(s[0])) {
      ret=-1;
   } else {
      ret=s[0]-'0';
      int i=1;
      while (isdigit(s[i])) {
         ret=ret*10+s[i]-'0';
         i++;
      }
   }
   return ret;
}
```

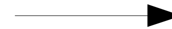
```
int get_value(char* s) {
   if (s==NULL) return -1;
   if (!isdigit(s[0])) return -1;
   int ret=s[0]-'0';
   int i=1;
   while (isdigit(s[i])) {
      ret=ret*10+s[i]-'0';
      i++;
   }
   return ret;
}
```

Esthétique des fonctions

- soigner la présentation:
  - commentaires
  - noms explicites
  - indentation
- regrouper les fonctions de même thème dans des .c

Passage d'adresse

```c
void add_3(int a) {
   a=a+3;
}

int main(int argc,char* argv[]) {
   int foo=14;
   add_3(foo);
   printf("foo=%d\n",foo);
   return 0;
}
```



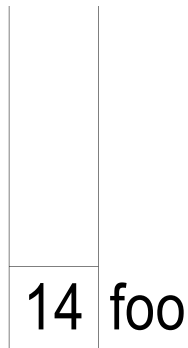
```
$>./a.out
foo=14
```

- pourquoi ?

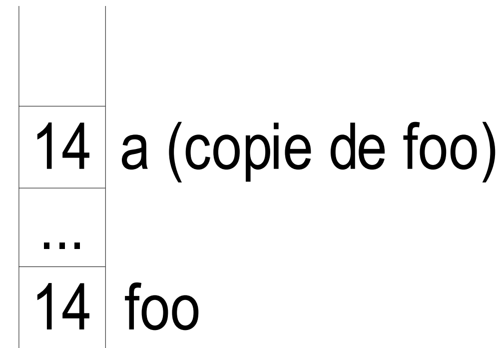
Passage d'adresse

- regardons la pile:

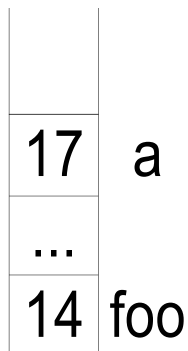
`int foo=14;`



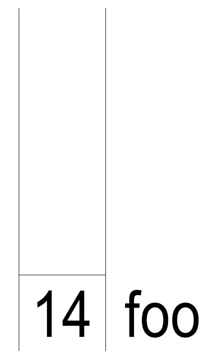
`add_3(foo);`



`a=a+3;`



`printf("foo=%d\n",foo);`



Passage d'adresse

- Comment faire ?

⇨ donner l'adresse de `foo` pour pouvoir modifier cette zone mémoire-là

- opérateur `&`:
  - `&abc` = adresse mémoire de la variable `abc`
  - comme dans `scanf`, qui a besoin de savoir où stocker les résultats de la saisie

Passage d'adresse

- pour indiquer qu'une fonction reçoit une adresse, on met `*` entre le type et le nom du paramètre:

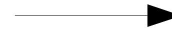
```
void copy(int a,int *b);
```

- `b` représente l'adresse d'un `int`
- `*b` est le contenu de la zone mémoire d'adresse `b`
- la taille de la zone dépend du type:

  `*a` différent dans `char *a` et `float *a`

Passage d'adresse

```
void add_3(int *a) {
   *a=*a+3;
}

int main(int argc,char* argv[]) {
   int foo=14;
   add_3(&foo);
   printf("foo=%d\n",foo);
   return 0;
}
```



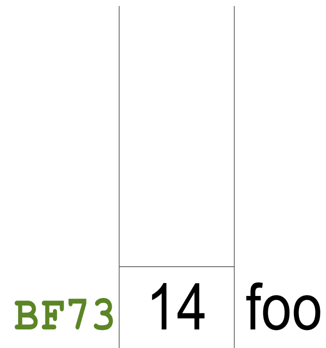
```
$>./a.out
foo=17
```

- ça marche!

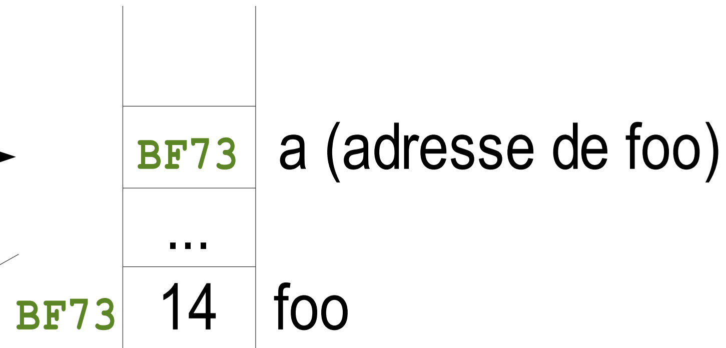
Passage d'adresse

- regardons la pile:

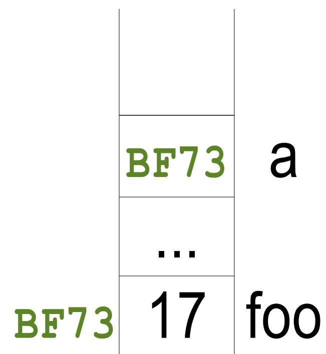
`int foo=14;`



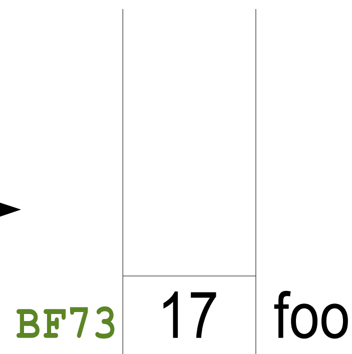
`add_3(&foo);`



`*a=*a+3;`



`printf("foo=%d\n",foo);`



Passage d'adresse

- utile:
  - quand on doit modifier un paramètre
  - quand on doit retourner plusieurs valeurs
  - quand on doit retourner une ou plusieurs valeurs + un code d'erreur

Modifier une variable

- exemple classique: les compteurs
- attention: **`*n++`** n'est pas **`(*n)++`** ⚡

```
/**
 * Reads from the given file the first
 * character that is not a new line. Returns
 * it or -1 if the end of file is reached. If
 * there are new lines, '*n' is updated.
 */
int read_char(FILE* f,int *n) {
   int c;
   while ((c=fgetc(f))!=EOF) {
      if (c=='\n') (*n)++;
      else return c;
   }
   return -1;
}
```

Retourner plusieurs valeurs

```c
/**
 * x/y is the irreducible fraction
 * so that x/y = a1/b1 + a2/b2
 */
void add_fractions(int a1,int b1,
                   int a2,int b2,
                   int *x,int *y) {
*x=a1*b2+a2*b1;
*y=b1*b2;
int p=gcd(*x,*y);
*x=*x/p;
*y=*y/p;
}
```

```c
/**
 * Returns the minimum of
 * the given array. Its
 * position is stored in '*pos'.
 */
float get_minimum(float t[],
                  int *pos) {
   int i;
   *pos=0;
   float min=t[0];
   for (i=1;i<N;i++) {
      if (t[i]<min) {
         min=t[i];
         *pos=i;
      }
   }
   return min;
}
```

Avec un code d'erreur

- on choisit plutôt de retourner le code d'erreur et de passer par adresse des variables pour stocker les résultats
- exemple: **scanf**

```
int main(int argc,char* argv[]) {
    int a,b,c,n;
    n=scanf("%d %d %d",&a,&b,&c);
    printf("%d %d %d %d\n",n,a,b,c);
    return 0;
}
```

```
$>./a.out
4 8 hello
2 4 8 4198592
```

Passage d'adresse

- les tableaux sont implicitement passés par adresse
- tableau = adresse de son premier élément

```c
/* équivalent à:
void foo(int* t) */
void foo(int t[]) {
   t[1]=666;
}

int main(int argc,char* argv[]) {
   int t[3];
   t[0]=t[1]=t[2]=0;
   foo(t);
   printf("%d\n",t[1]);
   return 0;
}
```

```
$>./a.out
666
```

Passage par adresse

- pas de ≠ formelle entre un *tableau de trucs* et un *truc* passé par adresse
- pour le compilateur, ces 3 fonctions ont le même prototype:

```
void foo(char s[]) {
   printf("(%s)\n",s);
}
```

```
void foo(char* s) {
   printf("(%s)\n",s);
}
```

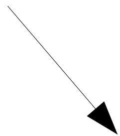
```
void foo(char *c) {
   *c='$';
}
```

- une notation pour s'y retrouver:

  `float* s` : `s` est un tableau de `float`

  `float *s` : `*s` est un `float` passé par adresse

Retourner une adresse

- ne jamais retourner l'adresse d'une variable 

```
$>gcc -Wall -ansi var.c
var.c: Dans la fonction « uppercase »:
var.c:14: AVERTISSEMENT: fonction
retourne l'adresse d'une variable
locale
$> ./a.out
¨÷ÿ¿Õí·
```

```
char* uppercase(char s[]) {
    char tmp[MAX];
    int i=-1;
    do {
        i++;
        tmp[i]=toupper(s[i]);
    } while (tmp[i]!='\0');
    return tmp;
}

int main(int argc,char* argv[]) {
    char* x=uppercase("abc");
    printf("%s\n",x);
    return 0;
}
```

`ABC` est sur la pile, et se fait écraser lors de l'appel à `printf`

Retourner une adresse

- conséquence:
  - on ne peut pas retourner un tableau
  - il faudra utiliser de l'allocation dynamique

Tableaux et pile

- tableau en variable locale = place occupée sur la pile
- attention aux explosions:

```
#define N 10000

void test_matrix() {
   int t[N][N];
   t[N-1][N-1]=17;
   printf("%d\n",t[N-1][N-1]);
}

int main(int argc,char* argv[]) {
   test_matrix();
   return 0;
}
```

→

```
$>./a.out
Segmentation fault
```

Fonction récursive

- fonction s'appelant elle-même:

```
void hanoi(int n,char src,
           char tmp,char dest) {
   if (n==1) {
      printf("%c --> %c\n",src,dest);
      return;
   }
   hanoi(n-1,src,dest,tmp);
   printf("%c --> %c\n",src,dest);
   hanoi(n-1,tmp,src,dest);
}

int main(int argc,char* argv[]) {
   hanoi(3,'A','B','C');
   return 0;
}
```

```
$>./a.out
A --> C
A --> B
C --> B
A --> C
B --> A
B --> C
A --> C
```

Fonction récursive

- attention à la condition d'arrêt
- fait gonfler la pile
- à éviter autant que possible:

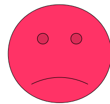
```
int factorial(int n) {
    if (n==1) return 1;
    return n*factorial(n-1);
}
```

```
int factorial(int n) {
    int r=1;
    while (n>1) {
        r=r*n--;
    }
    return r;
}
```

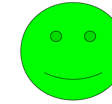
- utilisation judicieuse: cf cours d'algo

Fonction récursive

- si possible, ne tester qu'une seule fois les cas d'erreur



```
int sum(Tree* t,int *s) {
   if (t==NULL) return ERROR;
   *s=*s+t->value;
   if (t->left!=NULL) {
      sum(t->left,s);
   }
   if (t->right!=NULL) {
      sum(t->right,s);
   }
   return OK;
}
```



```
int sum_(Tree* t) {
   if (t==NULL) return 0;
   return t->value+sum_(t->left)
                  +sum_(t->right);
}

int sum(Tree* t,int *s) {
   if (t==NULL) return ERROR;
   *s=sum_(t);
   return OK;
}
```

Fonctions sur les chaînes

- fonctions définies dans `string.h`

```
char* strcpy(char* dest,const char* src);
char* strcat(char* dest,const char* src);
size_t strlen(const char* s);
```

- `strcpy` copie, `strcat` concatène, `strlen` mesure
- pas de test sur la validité des chaînes
  - ne doivent pas être `NULL`
  - `dest` doit être assez large

Fonctions sur les chaînes

```c
void get_email(char* first,
               char* last,
               char* email) {
strcpy(email,first);
strcat(email,".");
strcat(email,last);
strcat(email,"@univ-mlv.fr");
}
```

```
$>./a.out John Doe
John.Doe@univ-mlv.fr
```

```c
#define N 20

void tile(char* pattern,char* res) {
   int i,n=(N-1)/strlen(pattern);
   res[0]='\0';
   for (i=0;i<n;i++) {
      strcat(res,pattern);
   }
}

int main(int argc,char* argv[]) {
   char t[N];
   tile("pencil",t);
   printf("%s\n",t);
   return 0;
}
```

```
$>./a.out
pencilpencilpencil
```

Comparer des chaînes

- `int strcmp(const char* a,const char* b);`
- renvoie:

  0 si a=b

  <0 si a<b

  >0 si a>b

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

int main(int argc,char* argv[]) {
   printf("%d %d %d %d\n"
       ,strcmp("ball","ball")
       ,strcmp("ace","ball")
       ,strcmp("basic","ball")
       ,strcmp("ball","balls"));

   return 0;
}
```

```
$>./a.out
0 -1 1 -1
```

Fonctions mathématiques

- définies dans **math.h**:

```
double cos(double x);       double sin(double x);
double tan(double x);       double sqrt(double x);
double exp(double x);       double log(double x);
double log10(double x);     double pow(double x,double y);
```

- compiler avec l'option **-lm**

```
$>gcc -Wall -ansi math.c
/home/igm/paumier/tmp/ccTs2vJ2.o: dans la fonction « main »:
/home/igm/paumier/tmp/ccTs2vJ2.o(.text+0x20): référence
indéfinie vers « cos »
collect2: ld a retourné 1 code d'état d'exécution
$>gcc -Wall -ansi math.c -lm
```

Nombres aléatoires

- fonctions définies dans `stdlib.h`
- `int rand();`
  - donne un *n* entre `0` et `RAND_MAX`
- `void srand(unsigned int seed);`
  - initialise le générateur avec une graine
- pour une graine qui change:

  `time_t time(time_t*);` dans `time.h`

Nombres aléatoires

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

int main(int argc,char* argv[]) {
   int i;
   srand(time(NULL));
   for (i=0;i<10;i++) {
      printf("%d ",rand()%20);
   }
   printf("\n");
   return 0;
}
```

```
$>./a.out
0 10 5 5 9 5 7 6 7 11

$>./a.out
7 19 5 16 11 18 12 0 2 15
```