EXPRESSION

1) expression primaire

identificateur	`a b ca`
constante	`65 65.0 6.5e1 'A'`
chaîne de caractères	`"bonjour" "A" "a=%d\nb=%d\n"`
(*expression*)	`(a+2)`

2) opérateurs unaires postfixes

[*expression*]	`f[3] ce[1][2] f[a+3]`
(*liste d'expressions*)	`sqrt(a+3) printf("%d",i)`
.*identificateur*	`ca.re (*cd).im`
->*identificateur*	`cd->im`
`++` postincrémentation	`a++ d[2]++`
`--` postdécrémentation	`b--`

3) opérateurs unaires préfixes

`++` préincrémentation	`++a`
`--` prédécrémentation	`--a`
`&` adresse	`&a`
`*` indirection	`*cd *f **ce`
`+` identité	`+1`
`-` opposé (complément à 2)	`-3 -a -cd->re`
`~` complément à 1	`~0 ~a`
`!` non logique	`!2 !a !cd !cd->im`
(*type*)	`(int*)malloc(4) (float)a (int)ca.re`

4) opérateurs multiplicatifs

`*` produit	`3*5 3*a 5*(a+5)`
`/` quotient	`12/5 12.0/5 (a+3)/2.0`
`%` reste	`12%5 a*7%10 a++%4`

5) opérateurs additifs

`+` somme	`2+3 a+2 t+3 3+t`
`-` différence	`3*a-b &t[10]-&t[5] t+10-4`

6) opérateurs de décalage

`>>` décalage à droite	`1+4>>1 a*5>>2`
`<<` décalage à gauche	`34<<2`

7) opérateurs relationnels	`<= >= < > a<12`
8) opérateurs d'égalité	`== != a==b b!=2*c`
9) opérateurs min et max (`g++`)	`<? >? 3<?4 5>?7`
10) opérateur et bit à bit `&`	`a&1 12&5 a&~1`
11) opérateur ou exclusif bit à bit `^`	`12^5 a^1`
12) opérateur ou inclusif bit à bit `|`	`12|1 a|1`
13) opérateur et logique `&&`	`cd && cd->re`
14) opérateur ou logique `||`	`!cd || !cd->re`
15) opérateur conditionnel `? :`	`a==1?3:a+1`

expr. logique `?` *expr. quelconque* `:` *expr. conditionnelle*

16) affectations (de droite à gauche)	`= *= /= %= += -= <<= >>= &= ^= |=`
	`a=2 b+=3 a=b=c=0 a+=b+=c`
17) opérateur séquentiel `,`	`2,3 a=b,c`

INSTRUCTION

{*déclarations de variable instruction instruction instruction ...*}

`if(`*expression*`)` *instruction* [`else` *instruction*]

`switch(`*expression*`)` {*suite de [instruction / case constante*: */* `default:`*]* }

`for(`*[expression]*`;`*[expression]*`;`*[expression]*`)` *instruction*

`do` *instruction* `while (`*expression*`);`	`;`	*expression*`;`
`while (`*expression*`)` *instruction*	`break;`	`goto` *etiquette*`;`
`return` *expression*`;`	`continue;`	`etiquette:`*instruction*

Exemple de déclarations

```
int a, b, c=2, d[4], e[5]={1,5,4}, f[]={1,5,7,3}, *g, *h=e+2,
    *t[]={&a,d,e+1,f,f+3}, u[4][3], v[3][2]={{1,4},{2},{5,1}};
struct {int n; float x,y;} sa, sb, *sc, sd[4];
struct complexe {float re,im;} ca, cb[3], cc={1.3,4.5}, *cd;
struct complexe ce[3][5], cf;
struct fraction {int num,den;};
struct fraction fa, fb, fc[4]={{1,2},{4,3}};
typedef struct fraction frac, *pfrac, tfrac4[4];
frac fd, fe;
pfrac pfa, pfb=&fd;
tfrac4 tf1, tf2={{1,2},{3,4}};
typedef struct {int val; struct Chainon *suite;} Chainon, *Liste;
typedef struct chainon chainon, *liste;
struct chainon {int val; liste suite;};
typedef int tf(int,int);
```

Expressions et boucles

Qu'affiche le programme suivant ? (Les macros `t`, `T` et `F` calculent et affichent la valeur de leurs arguments.)

```
#include<stdio.h>
void aff(int t[],int n)
{ int i;
  for(i=0;i<n;++i) printf("%d ",t[i]);
  printf("\n");
}
int nbbit(int a) {int n=0; while(a) ++n,a&=a-1; return n;}
float ex(float x) {int i;float s,t;
  for(i=0,s=t=1;s!=s+t;) s+=t*=x/++i; return s;}
int p(int a,int b) {while(b&&(a%=b)) b%=a; return a+b;}
#define t(a) printf("%18s%4d  ",#a,a)
#define T(a,b,c,d) t(a),t(b),t(c),t(d),printf("\n")
#define F(abc...) printf("\n%f  %f  %f\n",abc)
int main()
{ int i,j,a[5],b[7];
  T(    (23,12),      (2+2,3*4),     (1,23%4?2,3:4,5),   1423^13^1423       );
  T(    23&12,        23&&12,        23|12,              23||12             );
  T(    23^12,        23==12,        23!=12,             23<12              );
  T(    3>>4,         3<<4,          3+2<<1+1,           3+2>>1+1           );
  T(    !23,          ~23,           !!23,               ~~23               );
  T(    ~(~23+~4),    nbbit(25),     (i=23, nbbit(i--)), (i=23, nbbit(--i))  );
  T(    p(35,91),     (i=23, ++i),   (i=23, i++),        (i=23, i+1, i)     );
  t(    (i=23, j=i++>>2, i+=++j, j+=i++)          );
  t(    (i=3, j=i>2&&i++>4&&i++>6?i+=32,i-=7:--i)  );
  F(    ex(1),        ex(0),         ex(-1)        );
  for(i=5;i--;) a[i]=i*i;
  aff(a,5);
  aff(a+1,3);
  for(i=0;i<7;aff(b,++i)) for(b[j=i]=1;--j>0;) b[j]+=b[j-1];
  return 0;
}
```

indications

Etudions l'effet de `a&=a-1`. Si `a` vaut par exemple 110111001000 (en binaire) alors on a :

```
a       110111001000
a-1     110111000111
a-1&a   110111000000
```

`a-1` s'obtient en remplaçant le 1 le plus à droite dans `a` par un 0, et tous les 0 à sa droite par des 1. Donc `a&=a-1` a pour effet de remplacer le 1 le plus à droite de `a` par un 0. A chaque itération de la boucle `while(a) ++n,a&=a-1;` un 1 de `a` est remplacé par un 0, la boucle est répétée jusqu'à ce que `a` ne contienne plus que des 0, et `n` compte les itérations. Donc `nbbit(a)` est le nombre de 1 dans l'écriture en binaire du nombre `a`.

Lors de l'évaluation de `ex(x)`, les valeurs succcessives des variables `i`, `s` et `t` en début de boucle sont :

`i`	`t`	`s`
0	1	1
1	x	$1+x$
2	$\frac{x^2}{2}$	$1+x+\frac{x^2}{2}$
3	$\frac{x^3}{6}$	$1+x+\frac{x^2}{2}+\frac{x^3}{6}$
n	$\frac{x^n}{n!}$	$1+x+\frac{x^2}{2}+\frac{x^3}{6}+\cdots+\frac{x^n}{n!}$

La boucle s'arrête quand `s+t` égale `s` c'est-à-dire quand `t` est devenu tellement petit que, compte tenu des arrondis, `s` ne change pas quand on lui ajoute `t`. Autrement dit `t` est négligeable devant `s`. La valeur de `ex(x)` est donc $\sum_{i=0}^{\infty}\frac{x^i}{i!}=e^x$. Lorsque x est positif, le calcul ne fait intervenir que des sommes de nombres positifs. Il n'y donc pas de pertes de précision. Pour x négatif on peut améliorer beaucoup la précision et un peu le temps de calcul en définissant `ex(x)` comme `1/ex(-x)`. Pour x grand, le nombre d'itérations est en $\Theta(x)$ (plus précisément $x+O(\sqrt{x})$). On peut donc nettement diminuer le temps de calcul ($\Theta(\ln x)$) en définissant `ex(x)` comme le carré de `ex(x/2)`.

```
float ex(float x)
{ int i; float s,t;
  if(x<0) s=1/ex(-x); else
  if(x>1) s=ex(x/2), s*=s; else
  for(i=0,s=t=1;s!=s+t;) s+=t*=x/++i;
  return s;
}
```

La double boucle

```
  for(i=0;i<7;aff(b,++i)) for(b[j=i]=1;--j>0;) b[j]+=b[j-1];
```

peut s'écrire de façon moins compacte

```
  for(i=0;i<7;++i)
  { b[i]=1;
    for(j=i-1;j>0;--j) b[j]+=b[j-1];
    aff(b,i+1);
  }
```

La procédure `aff` peut s'écrire de diverses façons :

```
void aff(int t[],int n) {int i; for(i=0;i<n;++i) printf("%d ",t[i]); printf ("\n");}
void aff(int *t ,int n) {int i; for(i=0;i<n;++i) printf("%d ",t[i]); putchar('\n');}
void aff(int *t ,int n) {int i; for(i=0;i<n;++i) printf("%d ",*t++); putchar('\n');}
void aff(int *t ,int n) {       for(   ; n ;--n) printf("%d ",*t++); putchar('\n');}
void aff(int *t ,int n) {       for(   ;n--;   ) printf("%d ",*t++); putchar('\n');}
void aff(int *t ,int n) {       for(   ;n--;++t) printf("%d ",*t  ); putchar('\n');}
void aff(int *t ,int n) {       while(  n--    ) printf("%d ",*t++); putchar('\n');}
```

Premiers programmes

```
#include<stdio.h>
int main()
{ int i=3,j=4;
  printf("%d+%d=%d\n",i,j,i+j);
  return 0;
}
#include<stdio.h>
int pgcd(int a,int b) {return b ?   pgcd(b,a%b) : a;}
int ppcm(int a,int b) {return b ? a/pgcd(a,b)*b : 0;}
int main()
{ int i,j;
  printf("Donnez deux nombres : ");
  scanf("%d%d",&i,&j);
  printf("%d+%d=%d\n",i,j,i+j);
  printf("%d-%d=%d\n",i,j,i-j);
  printf("%d*%d=%d\n",i,j,i*j);
  printf("%d/%d=%d\n",i,j,i/j);
  printf("%d%%%d=%d\n",i,j,i%j);
  printf("max(%d,%d)=%d\n",i,j,i<j?j:i);
  printf("min(%d,%d)=%d\n",i,j,i>j?j:i);
  printf("pgcd(%d,%d)=%d\n",i,j,pgcd(i,j));
  printf("ppcm(%d,%d)=%d\n",i,j,ppcm(i,j));
  return 0;
}
```

Arbres binaires

Voici des procédures qui, affiche un arbre binaire, crée un noeud, libère la place occupée par tous les noeuds d'un arbre :

```
#include<stdio.h>
#include<stdlib.h>
typedef struct noeud *arbre;
typedef struct noeud {int val; arbre fg,fd;} noeud;
arbre nouveau(arbre fg,int val,arbre fd)
{ arbre a=malloc(sizeof(noeud));
  a->val=v;
  a->fg=fg;
  a->fd=fd;
  return a;
}
void libere(arbre a) {if(a) libere(a->fg),libere(a->fd),free(a);}
void aff   (arbre a) {if(a) aff(a->fg),printf("%d ",a->val),aff(a->fd);}
int main()
{ arbre a;
  a=nouveau(nouveau(nouveau(0,3,0),4,0),5,nouveau(0,7,0));
  aff(a);
  libere(a);
  return 0;
}
```

Invariant de boucle

Compléter la procédure :

```
float puiss(float x,int n)
{ float y=1;   // A=x^n y est invariant tout au long de la procédure
  if(n<0) n=-n,...;
  while(n) if(n&1) n--,...;
           else    n/=2,...;
  return y;
}
```

Prouvez en ajoutant des assertions que chacune des deux versions de la fonction suivante retourne le contenu d'un tableau sur place.

```
void retourne(int t[],int n)
{ int i,j,x;
  for(i=0,j=n-1;i<j;++i,--j) x=t[i],t[i]=t[j],t[j]=x;
}
void retourne(int t[],int n)
{ int *u,*v,x;
  for(u=t,v=t+n-1;u<v;++u,--v) x=*u,*u=*v,*v=x;
}
```

Prouvez en ajoutant des assertions que chacune des deux versions de la procédure suivante trie un tableau sur place.

```
void tri(int t[],int n)
{ int i,x;
  if(n>1)
  { for(i=0;i<n-1;++i) if(t[i]>t[i+1])
    x=t[i],t[i]=t[i+1],t[i+1]=x;
    tri(t,n-1);
  }
}
void tri(int t[],int n)
{ int i,x;
  while(--n>0)
  for(i=0;i<n;++i) if(t[i]>t[i+1])
  x=t[i],t[i]=t[i+1],t[i+1]=x;
}
```

Mettre tri et retourne dans le programme suivant :

```
#include<stdio.h>
void aff (int t[],int n) {while(n--) printf("%d ",*t++); printf("\n");}
void lire(int t[],int n) {while(n--) scanf ("%d" , t++);              }
void tri (int t[],int n) ...
void retourne(int t[],int n) ...
int sum(int t[],int n) {return  t[0]+t[1]+...+t[n-1];}
#define n 30
int main()
{ int t[n],i;
  for(i=0;i<n;++i) t[i]=i*7%n;    aff(t,n);
  printf("%d+%d=%d\n",sum(t,n/2),sum(t+n/2,n-n/2),sum(t,n));
  retourne(t,n);                  aff(t,n);
  tri(t,n);                       aff(t,n);
  retourne(t,n);                  aff(t,n);
  i=getchar();                    return 0;
}
```

Modifier ce programme pour qu'il lise le tableau à trier.

Passage d'argument par référence (C++) et par pointeur (C)

```
void sompro(int a,int b,int &s,int &p){ s=a+b,p=a*b;}
int main()
{ int s,p;
  sompro(3,4,s,p);
  printf("%d %d\n",s,p);
  return 0;
}
void sompro(int a,int b,int *s,int *p){ *s=a+b,*p=a*b;}
int main()
{ int s,p;
  sompro(3,4,&s,&p);
  printf("%d %d\n",s,p);
  return 0;
}
void pgcd(int a,int b, int&c) {if(b)  c=a; else pgcd(b,a%b,c);}
void pgcd(int a,int b, int*c) {if(b) *c=a; else pgcd(b,a%b,c);}
liste enleve_dernier(liste &l)
{ if (l->suite) return enleve_dernier(l->suite);
  { liste d=l; l=0; return d;}
}
liste enleve_dernier(liste *l)
{ if ((*l)->suite) return enleve_dernier(&(*l)->suite);
  { liste d=*l; *l=0; return d;}
}
liste enleve_dernier(liste *l)
{ while ((*l)->suite) l=&(*l)->suite;
  { liste d=*l; *l=0; return d;}
}
```