Les assertions

Erreur utilisateur

Vous connaissez déjà les erreurs utilisateurs (saisies protégées)

```
int main()
{
    puts("Donnez un nombre entre 1 et 5");

    int nbr=0;
    int saisie=0;
    do
    {
        scanf("%d",&nbr);
        if(nbr<1 || nbr>5)
            puts("Ce nombre n'est pas correct");
        else
            saisie=1;
    }while(saisie!=1);

    return 0;
}
```

Dialogue avec l'utilisateur

Erreur programmeurs

Les erreurs utilisateurs ne sont pas les + courantes

=> Erreurs du programmeurs

```
int accede(int T[5],int k)
{
    return T[k];
}

int main()
{
    int T[5]={1,4,7,8,9};
    int valeur_1=accede(T,2);
    int valeur_2=accede(T,8);

    return 0;
}
```

Ne devrait accepter que des nombres entre [0,4]

Erreur programmeurs

```
int main()
{
    int T[5]={1,4,7,8,9};
    int valeur_1=accede(T,2);
    int valeur_2=accede(T,8);

    return 0;
}
```

```
int accede(int T[5],int k)
{
    if(k<0 || k>4)
    {
        puts("Probleme indice");
        abort();
    }
    return T[k];
}
```

Prévient de l'erreur du programmeur

Erreur programmeurs

```c
int main()
{
    int T[5]={1,4,7,8,9};
    int valeur_1=accede(T,2);
    int valeur_2=accede(T,8);

    return 0;
}
```

```c
int accede(int T[5],int k)
{
    if(k<0 || k>4)
    {
        puts("Probleme indice");
        abort();
    }
    return T[k];
}
```

Prévient de l'erreur du programmeur

Mais

3 lignes de debug pour une ligne d'action

Pas d'information sur l'origine de l'erreur
- fonction?
- fichier?
- quel indice?

Erreur programmeurs

```c
int main()
{
    int T[5]={1,4,7,8,9};
    int valeur_1=accede(T,2);
    int valeur_2=accede(T,8);

    return 0;
}
```

```c
int accede(int T[5],int k)
{
    assert(k>=0 && k<=5);
    return T[k];
}
```

+ Court
+ Précis

assert(condition)

<=> certifie que (condition est vraie)

Erreur programmeurs

```c
int main()
{
    int T[5]={1,4,7,8,9};
    int valeur_1=accede(T,2);
    int valeur_2=accede(T,8);

    return 0;
}
```

```c
int accede(int T[5],int k)
{
    if(k<0 || k>4)
    {
        puts("Probleme indice");
        printf("indice %d n'est pas dans [0,4]\n",k);
        puts("Erreur dans fonction accede");
        puts("fichier erreur.c, ligne 11\n");
        abort();
    }
    return T[k];
}
```

Précis

Ingérable pour un grand code

Erreur programmeurs

```c
int main()
{
    int T[5]={1,4,7,8,9};
    int valeur_1=accede(T,2);
    int valeur_2=accede(T,8);

    return 0;
}
```

```c
int accede(int T[5],int k)
{
    assert(k>=0 && k<=5);
    return T[k];
}
```

+ Court
+ Précis

```
erreur.c:8: accede: Assertion `k>=0 && k<=5' failed.
```

fichier ligne fonction condition

Fonction Assert

assert(*condition*)

- Certification d'un paramètre du code
- Détection d'erreur de code
- Aide pour le programmeur

Fonction Assert

assert(*condition*)

Condition uniquement vérifiée
en mode "developement"

gcc mon_pgm.c ⟶ assert vérifié

gcc mon_pgm.c -DNDEBUG ⟶ assert non vérifié

+ Pas de perte de temps: ne vous limitez pas sur les assert()

- Non utilisable pour les erreurs "runtime"
  > Utiliser les if

Fonction Assert

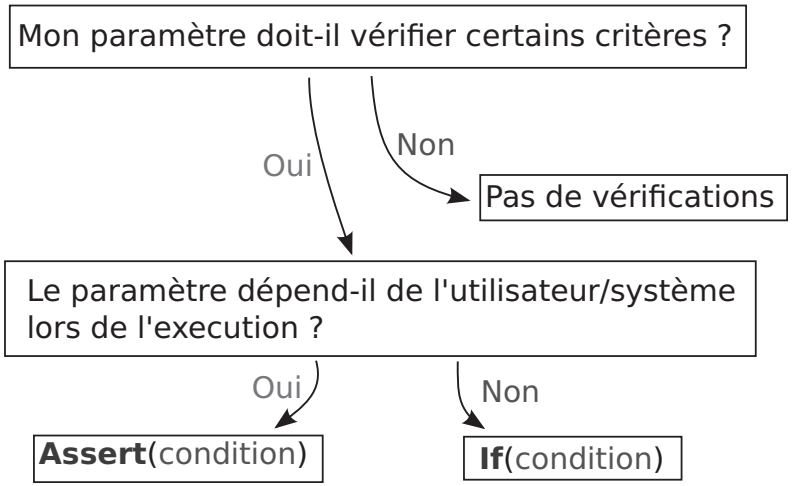
assert(*condition*)

Condition uniquement vérifiée
en mode "developement"

gcc mon_pgm.c ⟶ assert vérifié

gcc mon_pgm.c -DNDEBUG ⟶ assert non vérifié

Utilisation Assert



Méthodologies de conception

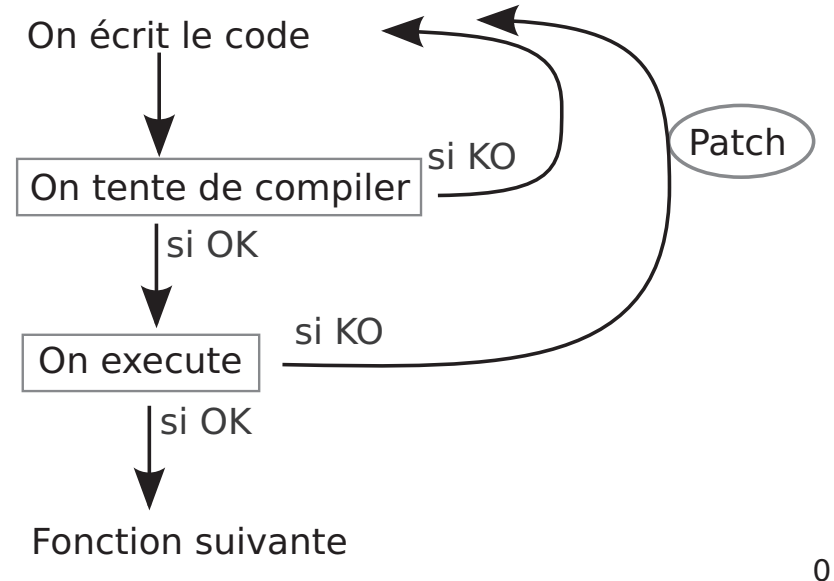
Développement par contrat
Développement par algorithmes
Développement par tests

Comment écrire un bon code ?

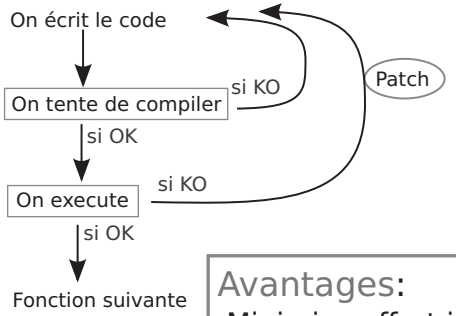
Plusieurs méthodologies de conception:

- Developpement par **contrats**
- Developpement par **tests**
- Developpement par **algorithmes**
- Developpement par hack

Developpement par hack



Developpement par hack



Avantages:
Minimise effort intellectuel à court terme

Inconvénients:
Patch dès le début
=> Difficile à lire (machine qui guide)
=> Ingérable pour des gros projets

Méthodologies de conception

→ **Développement par contrat**
Développement par algorithmes
Développement par tests

Developpement par contrats

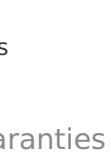
Avant d'écrire l'implémentation d'une fonction

On défini:

contrat avec le programmeur

1- Les **suppositions** sur les arguments d'entrées (ou état d'un système)

2- Les **certifications** après execution de la fonction



Developpement par contrats

Avant d'écrire l'implémentation d'une fonction

On défini:

contrat avec le programmeur

1- Les **suppositions** sur les arguments d'entrées (ou état d'un système)

2- Les **certifications** après execution de la fonction

Avantages:
- On empèche les cas particuliers/effets de bords
=> on le les oublies pas
- Auto-documente la fonction
- Aide au tests, garantie la validité
=> réduction des bugs

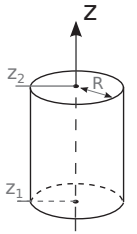
Inconvénients:
- Travail supplémentaire en amont

Developpement par contrats

Exemple:


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

//Calcul le volume du cylindre de rayon R entre z=[z1,z2]
float volume(float z1,float z2,float R)
{
  return   M_PI*R*R*(z2-z1);
}

int main()
{
  float V1=volume(1,2,0.5);
  float V2=volume(0,10,8);
  float V3=volume(-4,8,1);
  float V4=volume(5,1,1);
  float V5=volume(1,4,-1);

  return 0;
}
```

Fonction OK?



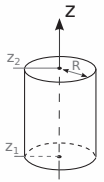
Developpement par contrats

Exemple:

```
/*Calcul le volume du cylindre de rayon R entre z=[z1,z2]
 * Le volume est obtenu par la formule V=  pi (z2-z1) R*R
 *
 * Necessite:
 *  - deux coordonnées flottantes (z1,z2) avec z1<z2
 *  - une coordonnée flottante R>0
 * Garantie:
 *  - renvoi un volume (flottant positif) correspondant au cylindre decrit.
 */
float volume(float z1,float z2,float R)
{
  assert(z2>z1);
  assert(R>0);

  float V=  M_PI*R*R*(z2-z1);

  assert(V>0);
  return V;
}
```

contrat
avec le programmeur

si non vérifiée:
quitte en indiquant la ligne

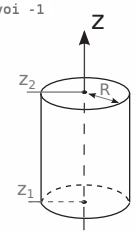


Developpement par contrats

Exemple:


```
/*Calcul le volume du cylindre de rayon R entre z=[z1,z2]
 * Le volume est obtenu par la formule V=  pi (z2-z1) R*R
 *
 * Necessite:
 *  - deux coordonnées flottantes (z1,z2) quelconques
 *  - une coordonnée flottante R quelconque
 * Garantie:
 *  - Si z1>z2 et R>0, renvoi un volume (flottant positif) correspondant au cylindre decrit.
 *  - Si z1=z2 ou R=0, renvoi zero
 *  - Si z1<z2 ou R<0, affiche une erreur en ligne de commande et renvoi -1
 */
float volume(float z1,float z2,float R)
{
  //cas du volume nul
  float epsilon=1e-5;
  if(fabs(z1-z2)<epsilon || fabs(R)<epsilon)
    return 0.0;
  //cas non geometrique
  if(z1<z2 || R<0)
    {printf("Erreur calcul volume\n");return -1.0;}
  //volume du cylindre
  float V=  M_PI*R*R*(z2-z1);
  return V;
}
```

autre contrat possible
avec le programmeur



Developpement par contrats

Synthèse:

+ Auto-documentation:
(le *code* est
la documentation)

```
/*Calcul le volume du cylindre de rayon R entre z=[z1,z2]
 * Le volume est obtenu par la formule V=2 pi (z2-z1) R.
 *
 * Necessite:
 *  - deux coordonnées flottantes (z1,z2) avec z1<z2
 *  - une coordonnée flottante R>0
 * Garantie:
 *  - renvoi un volume (flottant positif) correspondant au cylindre decrit.
 */
```

+ Cas particuliers gérés:
(pas d'oublis)

```
float V5=volume(1,4,-1);
```

- de debug
- explications, +lisible
=> gain de temps au final

Developpement par contrats

Exemple 2:


```
/* Calcul de la moyenne d'un ensemble de valeurs comprises entre 0 et 100
 * La moyenne est approxime sur l'entier le plus proche
 *
 * Necessite:
 *  - Un pointeur constant vers un ensemble de valeurs entieres. Pointeur non NULL.
 *    les valeurs du tableau sont comprises entre 0 et 100
 *  - Un entier positif indiquant la taille du tableau de donnees.
 * Garantie:
 *  - Renvoi la moyenne des valeurs du tableau (compris entre 0 et 100).
 *  - La moyenne est approximee au nombre entier le plus proche.
 */
int moyenne(const int* valeurs,unsigned int taille);
```

contrat
à mettre dans l'en-tête

```
int moyenne(const int* valeurs,unsigned int taille)
{
    assert(valeurs!=NULL);

    //somme de l'ensemble des valeurs
    int moyenne=0;
    int k=0;
    for(k=0;k<taille;++k)
      {
        int val=valeurs[k];
        assert(val>=0 && val<=100);
        moyenne += valeurs[k];
      }

    //passage en nombre flottants temporaires (virgules)
    float temporaire=(float)moyenne/taille;
    //approximation sur l'entier le plus proche
    moyenne=(int)(temporaire+0.5);

    return moyenne;
}
```

implémentation

Developpement par contrats

Proposition de syntaxe:

```
/**
 * Fonction NOM_FONCTION
 * ******************************
 *   DECRIT BUT DE LA FONCTION
 *
 *   Necessite:
 *     - DECRIT LES CONTRATS SUR LES VARIABLES ET ETAT DU SYSTEME
 *   Garantie:
 *     - DECRIT LES GARANTIES LORSQUE LES CONTRATS SONT RESPECTEES
 */
type_retour nom_fonction(type_var1 variable1,type_var2 variable2, ...);
```

A mettre dans le fichier d'en-tête : documentation

Programmation défensive?

Passer un contrat avec le programmeur?
Mais programmeur = vous?

Vous êtes la 1ère source d'erreur / bugs
Protégez vous contre vous même!

```
assert()
if(){}
...
```


Méthodologies de conception

Développement par contrat
→ **Développement par algorithmes**
Développement par tests

Programmation par algorithme

Avant d'écrire l'implémentation d'une fonction

On écrit:

L'**algorithme** dans un language humain

Attention: algorithme != code

Programmation par algorithme

Exemple:

```
/* Calcul de la moyenne d'un ensemble de valeurs comprises entre 0 et 100
 * La moyenne est approxime sur l'entier le plus proche
 */
int moyenne(const int* valeurs,unsigned int taille)
{
  //Algorithme: calcul de la moyenne
  //
  // Variable moyenne <- 0
  //
  // Pour toutes les valeurs v du tableau
  //   verifier v compris entre [0,100]
  //   ajouter v a moyenne
  // Fin Pour
  //
  // Diviser moyenne par la taille du tableau
  // Arrondir a l'entier le plus proche
  //
  // Renvoyer valeur de la moyenne
  //
}
```


Programmation par algorithme

Exemple:

```
/* Calcul de la moyenne d'un ensemble de valeurs comprises entre 0 et 100
 * La moyenne est approxime sur l'entier le plus proche
 */
int moyenne(const int* valeurs,unsigned int taille)
{
  //Algorithme: calcul de la moyenne
  //
  // Variable moyenne <- 0
  //
  // Pour toutes les valeurs v du tableau
  //   verifier v compris entre [0,100]
  //   ajouter v a moyenne
  // Fin Pour
  //
  // Diviser moyenne par la taille du tableau
  // Arrondir a l'entier le plus proche
  //
  // Renvoyer valeur de la moyenne
  //
}
```

complétion

```
/* Calcul de la moyenne d'un ensemble de valeurs comprises entre 0 et 100
 * La moyenne est approxime sur l'entier le plus proche
 */
int moyenne(const int* valeurs,unsigned int taille)
{
  assert(valeurs!=NULL);

  // Variable moyenne <- 0
  int moyenne=0;

  // Pour toutes les valeurs v du tableau
  unsigned int k=0;
  for(k=0;k<taille;++k)
    {
      //   verifier v compris entre [0,100]
      int v=valeurs[k];
      assert(v>=0 && v<=100);
      //   ajouter v a moyenne
      moyenne += v;
    }

  // Diviser moyenne par la taille du tableau
  float temporaire=(float)moyenne/taille;
  // Arrondir a l'entier le plus proche
  moyenne=(int)(temporaire+0.5);


  // Renvoyer valeur de la moyenne
  return moyenne;
}
```


Programmation par algorithme

Exemple:

```
void recherche_minimum(float x0,float y0)
{
    float x=x0;
    float y=y0;
    float e=1e-5;

    float dx[8]={e,e,0,-e,-e,-e,0,e};
    float dy[8]={0,e,e,e,0,-e,-e,-e};

    float v=fonction_inconnue(x,y);
    float min=v;
    int k_opt=-1;
    float n[8];

    unsigned int k=0;
    do
    {
        v=fonction_inconnue(x,y);
        min=v;
        k_opt=-1;
        for(k=0;k<8;++k)
        {
            n[k]=fonction_inconnue(x+dx[k],y+dy[k]);
            if(min>n[k])
            {
                min=n[k];
                k_opt=k;
            }
        }
        if(k_opt!=-1)
        {
            x+=dx[k_opt];
            y+=dy[k_opt];
        }
    }
    while(k_opt!=-1);

    printf("f(%f %f)=%f\n",x,y,v);
}
```

??

```
int main()
{
  recherche_minimum(1,2);
}

float fonction_inconnue(float x,float y)
{
  return exp(-cos(x-y)+x*x+y*y)+(0.5-sqrt(x*y))*exp(-x*x-0.5*y*y);
}
```


Programmation par algorithme

Exemple:

```
void recherche_minimum(float x0,float y0)
{
    //****************************************
    //Recherche de minimum de fonction_inconnue
    //****************************************
    //
    // Algorithme:
    //
    // Initialise (x,y) a (x0,y0)
    //
    // Pre-stocker la table de deplacement pour 8 voisins
    //   C'est a dire: deplacement[k]={ (-dx,+dy)[3] (  0,+dy)[2] (+dx,+dy)[1] }
    //                                 { (-dx,  0)[4]             (+dx,  0)[0] }
    //                                 { (-dx,-dy)[5] (  0,-dy)[6] (+dx,-dy)[7] }
    //
    // Tant que minimum non atteint
    //
    // Affecter a V0 la valeur de fonction_inconnue(x,y)
    // Pour tous les 8 voisins [k] de (x,y)
    //
    //   Soit (x2,y2) le voisin courant de (x,y)
    //     C'est a dire: x2=x +/- dx
    //                   y2=y +/- dy
    //
    //   Affecter a V[k] la valeur de fonction_inconnue(voisin courant)
    //
    // Fin Pour
    //
    // Cherchez le minimum de V[k] pour k=k_optimal
    //
    // Si V[k_optimal] plus petit que V0
    //   Alors avancer (x,y) dans la direction du voisin k_optimal
    // Fin Si
    //
    //
    // Fin Tant que
```

Programmation par algorithme

Exemple:

```
void recherche_minimum(float x0,float y0)
{
    //*****************************************
    //Recherche de minimum de fonction_inconnue
    //*****************************************
    //
    // Algorithme:
    //
    // Initialise (x,y) a (x0,y0)
    float x=x0;
    float y=y0;

    // Prestocker la table de deplacement pour 8 voisins
    //  C'est a dire: deplacement[k]={ (-dx,+dy)[3] (  0,+dy)[2] (+dx,+dy)[1] }
    //                                { (-dx,  0)[4]              (+dx,  0)[0] }
    //                                { (-dx,-dy)[5] (  0,-dy)[6] (+dx,-dy)[7] }
    float deplacement[8][2]={{dx,0},{dx,dy},{0,dy},{-dx,+dy},{-dx,0},{-dx,-dy},{0,-dy},{+dx,-dy}};

    //
    // Tant que minimum non atteint
    //
    int minimum_atteint=0;
    while(minimum_atteint==0)
    {
        // Affecter a V0 la valeur de fonction_inconnue(x,y)
        float V0=fonction_inconnue(x,y);

        // Pour tous les 8 voisins [k] de (x,y)
        float V[8];
        unsigned int k=0;
        for(k=0;k<8;++k)
        {
        //
        //   Soit (x2,y2) le voisin courant de (x,y)
        //      C'est a dire: x2=x +/- dx
        //                    y2=y +/- dy
        //
            float x2=x+deplacement[k][0];
            float y2=y+deplacement[k][1];
```

```
        //   Affecter a V[k] la valeur de fonction_inconnue(voisin courant)
        //

            V[k]=fonction_inconnue(x2,y2);
        // Fin Pour
        }
        //
        // Cherchez le minimum de V[k] pour k=k_optimal
        int k_optimal=trouve_min(V,8);

        //
        // Si V[k_optimal] plus petit que V0
        //   Alors avancer (x,y) dans la direction du voisin k_optimal
        // Sinon
        //   minimum_atteint est vrai
        // Fin Si

        if(V[k_optimal]<V0)
        {
            x += deplacement[k_optimal][0];
            y += deplacement[k_optimal][1];
        }
        else
        {
            minimum_atteint=1;
        }
        //
        //
        // Fin Tant que
    }
}
```

```
//trouve le minimum d'un vecteur
int trouve_min(float V[],int taille)
{
    assert(taille>=1);

    // Initialiser minimum au premier element du vecteur
    // Initialiser indice k_optimal a 0
    //
    // Pour tout les elements d'indice [k] restants du vecteur V
    // Si minimum > V[k]
    //   Alors affecter minimum=V[k],
    //         et sauvegarder k_optimal=k
    // Fin Si
    // Fin Pour
    //
    // Retourner k_optimal

    float minimum=V[0];
    int k_optimal=0;

    unsigned int k=0;
    for(k=1;k<taille;++k)
    {
        if(minimum>V[k])
        {
            minimum=V[k];
            k_optimal=k;
        }
    }

    return k_optimal;
}
```


Programmation par algorithme

Synthèse

+ Documentation du code automatique

+ Aide à l'écriture du code

+ lisible
+ code meilleur qualité

Programmation par algorithme

Ne pas confondre algorithme et code

ex. A NE PAS FAIRE!

```
int moyenne(const int* valeurs,unsigned int taille)
{
  //Algorithme: calcul de la moyenne
  //
  // Variable moyenne=0
  // Variable k=0
  // Pour k=0, tant que k<taille, k++
  //   assert valeurs[k]<0 && valeurs[k]<100
  //   moyenne += valeurs[k]
  // Fin Pour
  //
  // moyenne /= taille;
  // moyenne=(int)(moyenne+0.5)
  //
  // return moyenne
}
```

```
int moyenne(const int* valeurs,unsigned int taille)
{
  //Algorithme: calcul de la moyenne
  //
  // Variable moyenne=0
  int moyenne=0;
  // Variable k=0
  int k=0;
  // Pour k=0, tant que k<taille, k++
  for(k=0;k<taille;++k)
    {
    //   assert valeurs[k]<0 && valeurs[k]<100
       assert(valeurs[k]>=0 && valeurs[k]<=100);
       //   moyenne += valeurs[k]
       moyenne+=valeurs[k];
       // Fin Pour
    }
  //
  // moyenne /= taille;
  float temporaire=(float)moyenne/taille;
  // moyenne=(int)(moyenne+0.5)
  moyenne=(int)(temporaire+0.5);
  //
  // return moyenne
  return moyenne;
}
```

réécriture mots pour mots
les commentaires n'apportent aucune amélioration de lisibilité au code

Méthodologies de conception

Développement par contrat
Développement par algorithmes
→ **Développement par tests**

Programmation par tests

Avant d'écrire l'implémentation d'une fonction

On écrit:

Les tests que doivent satisfaire la fonction

En général:
On connait ce que doit faire la fonction
avant de connaitre son code

Programmation par tests

Exemple:

```
//1 valeur, attend 1
int t1[]={50};
if(moyenne(t1,sizeof(t1)/sizeof(int))!=50)
    ECHEC_TEST;

//arrondie au nombre superieur, attend 72
int t2[]={45,80,90};
if(moyenne(t2,sizeof(t2)/sizeof(int))!=72)
    ECHEC_TEST;

//arrondi au nombre inferieur, attend 71
int t3[]={45,80,89};
if(moyenne(t3,sizeof(t3)/sizeof(int))!=71)
    ECHEC_TEST;

//valeurs constantes, attend 45
int t4[]={45,45,45,45,45};
if(moyenne(t4,sizeof(t4)/sizeof(int))!=45)
    ECHEC_TEST;
```

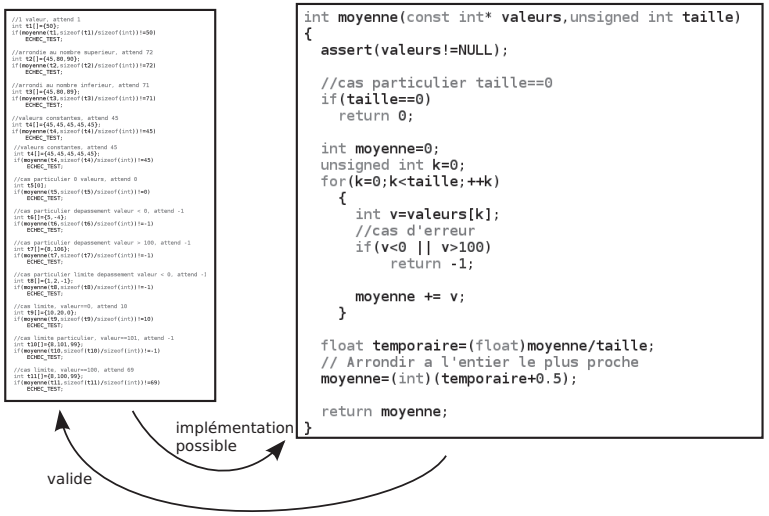
```
//valeurs constantes, attend 45
int t4[]={45,45,45,45,45};
if(moyenne(t4,sizeof(t4)/sizeof(int))!=45)
    ECHEC_TEST;

//cas particulier 0 valeurs, attend 0
int t5[0];
if(moyenne(t5,sizeof(t5)/sizeof(int))!=0)
    ECHEC_TEST;

//cas particulier depassement valeur < 0, attend -1
int t6[]={5,-4};
if(moyenne(t6,sizeof(t6)/sizeof(int))!=-1)
    ECHEC_TEST;

//cas particulier depassement valeur > 100, attend -1
int t7[]={8,106};
if(moyenne(t7,sizeof(t7)/sizeof(int))!=-1)
    ECHEC_TEST;

//cas particulier limite depassement valeur < 0, attend -1
int t8[]={1,2,-1};
if(moyenne(t8,sizeof(t8)/sizeof(int))!=-1)
    ECHEC_TEST;

//cas limite, valeur==0, attend 10
int t9[]={10,20,0};
if(moyenne(t9,sizeof(t9)/sizeof(int))!=10)
    ECHEC_TEST;

//cas limite particulier, valeur==101, attend -1
int t10[]={8,101,99};
if(moyenne(t10,sizeof(t10)/sizeof(int))!=-1)
    ECHEC_TEST;

//cas limite, valeur==100, attend 69
int t11[]={8,100,99};
if(moyenne(t11,sizeof(t11)/sizeof(int))!=69)
    ECHEC_TEST;
```

tests différents cas:
valides + invalides

fixe entrée
=> attend sortie spécifique

Programmation par tests

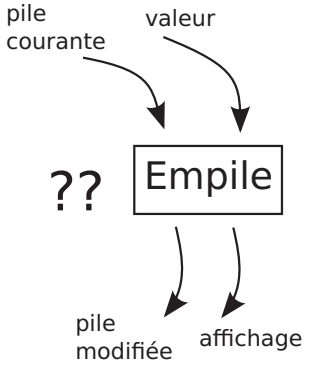
Exemple:



Programmation par tests

Exemple:

```
#define TAILLE_MAX 15

struct pile
{
  int indice;
  int buffer[TAILLE_MAX];
};

void initialise(struct pile* p)
{
  p->indice=0;
}
int depile(struct pile* p)
{
  if(p->indice>0)
    return p->buffer[--p->indice];
  printf("Erreur liste vide\n");
  return 0;
}
int est_vide(const struct pile* p)
{
  return p->indice==0;
}
```

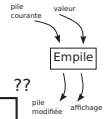


Programmation par tests

Exemple:

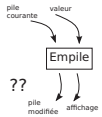
```
#define TAILLE_MAX 15

struct pile
{
  int indice;
  int buffer[TAILLE_MAX];
};

void initialise(struct pile* p)
{
  p->indice=0;
}
int depile(struct pile* p)
{
  if(p->indice>0)
    return p->buffer[--p->indice];
  printf("Erreur liste vide\n");
  return 0;
}
int est_vide(const struct pile* p)
{
  return p->indice==0;
}
```

```
//*********************//
//Test Pile
//*********************//
struct pile p;
initialise(&p);

empile(&p,5);
int var_0=depile(&p); //doit retourner 5
if(var_0!=5){printf("Erreur %d\n",__LINE__);}

empile(&p,6);
empile(&p,7);
int var_1=depile(&p); //doit retourner 7
int var_2=depile(&p); //doit retourner 6
if(var_1!=7 || var_2!=6){printf("Erreur %d\n",__LINE__);}

int var_3=est_vide(&p); //doit retourner vrai
if(var_3!=1) {printf("Erreur %d\n",__LINE__);}

int k=0;
for(k=0;k<TAILLE_MAX;k++)
  empile(&p,k);
empile(&p,5); //doit afficher une erreur: vecteur plein
int var_4=depile(&p); //doit retourner TAILLE_MAX-1
if(var_4!=TAILLE_MAX-1) {printf("Erreur %d\n",__LINE__);}
```



Programmation par tests

Exemple:




```
#define TAILLE_MAX 15

struct pile
{
  int indice;
  int buffer[TAILLE_MAX];
};

void initialise(struct pile* p)
{
  p->indice=0;
}
int depile(struct pile* p)
{
  if(p->indice>0)
    return p->buffer[--p->indice];
  printf("Erreur liste vide\n");
  return 0;
}
int est_vide(const struct pile* p)
{
  return p->indice==0;
}
```

```
//*********************//
//Test Pile
//*********************//
struct pile p;
initialise(&p);

empile(&p,5);
int var_0=depile(&p); //doit retourner 5
if(var_0!=5){printf("Erreur %d\n",__LINE__);}

empile(&p,6);
empile(&p,7);
int var_1=depile(&p); //doit retourner 7
int var_2=depile(&p); //doit retourner 6
if(var_1!=7 || var_2!=6){printf("Erreur %d\n",__LINE__);}

int var_3=est_vide(&p); //doit retourner vrai
if(var_3!=1) {printf("Erreur %d\n",__LINE__);}

int k=0;
for(k=0;k<TAILLE_MAX;k++)
  empile(&p,k);
empile(&p,5); //doit afficher une erreur: vecteur plein
int var_4=depile(&p); //doit retourner TAILLE_MAX-1
if(var_4!=TAILLE_MAX-1) {printf("Erreur %d\n",__LINE__);}
```

pile courante

valeur

```
void empile(struct pile* p,int a)
{
  if(p->indice<TAILLE_MAX)
    p->buffer[p->indice++]=a;
  else
    printf("Erreur buffer plein\n");
}
```

pile modifiée

affichage

Programmation par tests

Synthèse:

+ Aide à définir l'interface de la fonction (paramètres, entrées/sorties)
(pas d'ajout de paramètres une fois le corps écrit)

+ Gère les cas particuliers
(pas d'oublis)

+ Debug de la fonction immédiate: plus besoin de réécrire des tests
=> Assure la validité de la fonction

+ Gagne du temps sur l'écriture des tests
(qui doivent avoir lieu plus tard sinon)

+ Valider les fonctionnalitées à tout moment.
Code qui ne régresse pas

Organisation des données

Pointeurs/Structs

Declaration
Passage de paramètres
Mot clé **const**

Déclaration structs

explicite

```
struct v3
{
    float x;
    float y;
    float z;
};

int main()
{
    struct v3 mon_vecteur_3d;
}
```

struct anonyme

```
typedef struct
{
    float x;
    float y;
}v2;

int main()
{
    v2 mon_vecteur_2d;
}
```


Bonne pratique: documentation

Documentez un maximum vos déclarations de structs

```
//Structure contenant un vecteur de coordonnees 3D
struct v3
{
    float x; //coordonnee x
    float y; //coordonnee y
    float z; //coordonnee z
};
```

roles,
plage de variations,
valeurs typiques,
explications, ...


```
#define TAILLE_MAX 50 //taille maximal d'un nom

//Structure contenant les informations d'un etudiant
struct etudiant
{
    char nom[TAILLE_MAX]; //Nom de l'etudiant
    char prenom[TAILLE_MAX]; //Prenom de l'etudiant

    int classe;//0 correspond a 3ETI
               //1 correspond a 4ETI
               //2 correspond a 5ETI

    float moyenne;//sa moyenne
                  //nombre flottant a 3 decimales
                  //compris entre 0.0 et 20.0
};
```


Importance struct+fonctions

Importance des structs + fonctions

```
//Structure contenant un vecteur de coordonnees 3D
struct v3
{
    float x; //coordonnee x
    float y; //coordonnee y
    float z; //coordonnee z
};

//Affecte les coordoonees (x,y,z) au vecteur
void v3_set(struct v3* vec,float x,float y,float z);
//Renvoie la norme du vecteur
float v3_norm(const struct v3* vec);
//Affiche les coordonnees du vecteur en ligne de commande
void v3_print(const struct v3* vec);

int main()
{
    struct v3 mon_vecteur_3d;
    v3_set(&mon_vecteur_3d,1,2,2);
    float n=v3_norm(&mon_vecteur_3d);

    printf("||");
    v3_print(&mon_vecteur_3d);
    printf("||");
    printf("=%f\n",n);
}
```

fonctions
agissants sur une struct

Implémentation

```
void v3_set(struct v3* vec,float x,float y,float z)
{
    vec->x=x;
    vec->y=y;
    vec->z=z;
}
float v3_norm(const struct v3* vec)
{
    return sqrt(vec->x*vec->x+vec->y*vec->y+vec->z*vec->z);
}
void v3_print(const struct v3* vec)
{
    printf("(%f,%f,%f)",vec->x,vec->y,vec->z);
}
```


Importance struct+fonctions

Modélise une **abstraction** *vecteur*
voiture / roues, ...

\+ Cache la complexité
\+ Manipulation aisée des données

Organisation des données

Pointeurs/Structs

Declaration
→ **Passage de paramètres**
Mot clé **const**

Passage de paramètres

Passage de paramètres

```
int main()
{
    struct v3 vec_1;
    v3_set(&vec_1,1,2,2);

    struct v3 vec_2;
    vec_2=vec_1;
    vec_2.y=3;

    v3_print(&vec_1);
    v3_print(&vec_2);
}
```

copie

copie: struct a=b

```
Pour tout les champs  <k>
  a.<k>=b.<k>
Fin Pour
```


Passage de paramètres

```
void fonction(struct v3 a)
{
    a.x=8;
}

int main()
{
    struct v3 vec_1={1,1,5};
    fonction(vec_1);
    printf("%d",vec_1.x);
}
```

Passage par copie

Passage de paramètres

```
float v8_fonction(struct v8 a)
{
    return a.x0+a.x1;

}

int main()
{
    struct v8 vec_1={1,1,5,1,4,5,7,8};
    for(k=0;k<5000000;++k)
    {
        v8_fonction(vec_1);
    }
}
```

Copie réellement nécessaire ?

Passage de paramètres

```
float v8_fonction(struct v8* a)
{
    return a->x0+a->x1;

}

int main()
{
    struct v8 vec_1={1,1,5,1,4,5,7,8};
    for(k=0;k<5000000;++k)
    {
        v8_fonction(&vec_1);
    }
}
```

Passage par pointeur
= 8 octets < sizeof(v8)

32 octets

Passage de paramètres

Passage de paramètres

```
float v3_norm(struct v3* a);

int main()
{
    struct v3 vec_1={1,1,5};
    float n=v3_norm(&vec_1);
    printf("%f\n",n);
}
```

Pointeur
=>danger d'intégrité

Comment détecter/éviter?

```
float v8_norm(struct v3* a)
{
    float n=sqrt(a->x*a->x+a->y*a->y+a->z*a->z);
    a->x=152;
    return n;
}
```


Passage de paramètres

Mot clé **const**

```
float v3_norm(const struct v3* a);
```

Assure que a n'est pas modifié
Pas besoin de regarder l'implémentation

Passage de paramètres

Mot clé **const**

```
float v3_norm(const struct v3* a);
```

Assure que a n'est pas modifié
Pas besoin de regarder l'implémentation

implémentation identique

```
float v3_norm(const struct v3* a)
{
    float n=sqrt(a->x*a->x+a->y*a->y+a->z*a->z);
    return n;
}
```

Organisation des données

Pointeurs/Structs

Declaration
Passage de paramètres
→ **Mot clé const**

Mot clé **const**

En cas d'erreur:

```
float v3_norm(const struct v3* a)
{
    float n=sqrt(a->x*a->x+a->y*a->y+a->z*a->z);
    a->x=153;
    return n;
}
```

compilation

```
gcc struct.c -lm -g -lrt
struct.c: In function 'v3_norm':
struct.c:52:5: error: assignment of member 'x' in read-only object
```

Evite de mal coder!!

Mot clé **const:** utilisation

const se *propage*

```
struct CD_musique
{
    char titre[50];
    char auteur[50];
    int prix;
};
struct DVD_film
{
    char titre[50];
    int duree;
    int prix;
};
struct etagere
{
    struct CD_musique CD[100];
    struct DVD_film DVD[100];
};
```


Mot clé **const:** utilisation

const se *propage*


```
struct CD_musique
{
    char titre[50];
    char auteur[50];
    int prix;
};
struct DVD_film
{
    char titre[50];
    int duree;
    int prix;
};
struct etagere
{
    struct CD_musique CD[100];
    struct DVD_film DVD[100];
};
```


```
//renvoie pointeur vers CD sur mon etagere
CD_musique* retourne_CD_musique(etagere* mon_etagere,int indice)
{
    return &( mon_etagere->CD[indice] );
}
//renvoie pointeur (constant) vers CD sur mon etagere
const CD_musique* cherche_CD_musique(const etagere* mon_etagere,int indice)
{
    return &( mon_etagere->CD[indice] );
}
```

Mot clé **const:** utilisation

const se *propage*

```
//renvoie pointeur vers CD sur mon etagere
CD_musique* retourne_CD_musique(etagere* mon_etagere,int indice)
{
    return &( mon_etagere->CD[indice] );
}
//renvoie pointeur (constant) vers CD sur mon etagere
const CD_musique* cherche_CD_musique(const etagere* mon_etagere,int indice)
{
    return &( mon_etagere->CD[indice] );
}
```

```
struct CD_musique
{
    char titre[50];
    char auteur[50];
    int prix;
};
struct DVD_film
{
    char titre[50];
    int duree;
    int prix;
};
struct etagere
{
    struct CD_musique CD[100];
    struct DVD_film DVD[100];
};
```

```
int main()
{
    struct etagere mon_etagere;
    //rempli etagere ...

    struct CD_musique *mon_CD_1=retourne_CD_musique(&mon_etagere,3);
    mon_CD_1->prix=20;

    const struct CD_musique *mon_CD_2=cherche_CD_musique(&mon_etagere,3);
    printf("%d\n",mon_CD_2->prix);
}
```

modifiable

constant

Mot clé **const:** Synthèse

- Passer les structs par pointeurs (choix)

- Tous pointeurs passés en **const** ! (bonne programmation)

=> Enlever le const uniquement si nécessaire

```
char *filename="blabla";
const char* filename="blabla";

void ma_fonction(char *filename);
void ma_fonction(const char *filename);
```

vérification du compilateur code n'est pas impacté (aucun désavantage)

Mot clé **const:** placement

Identique:

const int a; int **const** a;	**const** int *a; int **const** *a;

Attention au placement de *

int **const** *a; int ***const** a;	ne sont pas équivalents!

Mot clé **const:** placement

- pointeur lui même constant
- pointeur vers valeur constante

```
char* const filename_1="fichier_1";
const char* filename_2="fichier_2";
```

```
char* const filename_1="fichier_1";
const char* filename_2="fichier_2";

filename_1=NULL; //erreur
filename_2=NULL; //OK
```

- on peut cumuler:

```
const char* const filename_3="mon_fichier";
```

- pointeurs mulitples:

```
int **p1;
int const **p2;
int const *const *p3;
int const *const *const p4;
int *const *const p5;
int **const p6;
int *const*p7;
```

=> Trop complexe
=> Utiliser des structs intermédiaires