Gestion des erreurs

Types d'erreurs
Erreur d'arrêt immédiats
Transmission d'information d'erreur
- retour d'arguments
- passage de paramètres

Erreur utilisateur

Gestion des erreurs

Problème complexe

```
void ouverture_fichier(const char* filename)
{
  FILE* fid=NULL;
  fid=fopen(filename,"r");
  if(fid==NULL)
    {
      //GESTION D'ERREUR
    }
}
```

Que faire?

* *Quitter*
* *Indiquer l'erreur*
* *Revenir à la fonction appelante*
* *Ecrire un log dans un fichier*
* *Ne rien faire*
* *Mettre une variable globale d'erreur à jour*
* *...*

Gestion des erreurs

```c
void ouverture_fichier(const char* filename)
{
  FILE* fid=NULL;
  fid=fopen(filename,"r");
  if(fid==NULL)
    {
      //GESTION D'ERREUR
    }
}
```

Constat: plusieurs solutions possibles: dépend du contexte

ex. Si le fichier est censé exister (programmation par contrat)
=> Erreur de programmation: il faut quitter et debugger

Si le fichier peut ne pas exister
=> Afficher ou non, et retenter potentiellement

travail de la fonction appelante
=> elle doit être mise au courant

Glossaire des types d'erreurs

1. **Erreur de programmation** (/bugs)

ex.
Non respect d'un contrat de programmation.
Indice de tableau <0 ou > taille
Ecriture sur un pointeur NULL, ou pointant sur une adresse non valide
...

Doit être détectée le plus tôt possible

rappel de la priorité

```
+++ Compilateur
++  Par tests unitaires
+   Par assertions
-   Par tests integrations
--  Par hasard par le programmeur
--- Par le client (catastrophe)
```

Que faire:

Aboutit à l'arrêt immédiat du programme
Message d'erreur à l'attention d'un programmeur:
Doit être suivit d'une modification du code

doit être précis sur
** la cause*
** endroit du code*

ex. printf("Erreur du a ... ligne %d, fonction %s, fichier %s\n",__LINE__,__FUNCTION__,__FILE__);

Note: Ne doit Jamais arriver en production/chez le client

Glossaire des types d'erreurs

2. Erreur système critique

ex.
Manque de place mémoire RAM/disque
(allocation mémoire qui échoue)
Operation importante non permise par le noyaux
Reception d'un signal de type "kill"
...

Généralement difficile à prévoir (dépend des conditions d'utilisations)
=> Bien vérifier les retours des appels systèmes

Que faire:

Aboutit généralement à l'arrêt du programme
(potentiellement après nettoyage: désallocation, suppression de fichiers temporaires, etc.)
Message d'erreur à l'attention de l'utilisateur (/potentiellement programmeur)

Note: Peut arriver chez le client

=> tests avec if{...} (et pas avec assertions)

(cas connu:
kernel panic / blue screen of death)

Glossaire des types d'erreurs

3. Erreur système exceptionnelles non critique

ex.
Ressource occupée (fichier, variable partagée, etc.)
Fichier de configuration manquant ou invalide
...

Note: L'aspect "exceptionnelle" et "critique" dépend de l'utilisation

Que faire:

Aboutit au traitement spécifique par la fonction appelante.
(temporisation, création de nouveaux fichiers, utilisation de valeurs par défauts, ...)
Message d'erreur à l'attention de l'utilisateur en mode debug
(pas forcément d'alerte en mode release si correctement traité).

ex.

```
char fichier[]="mon_fichier_de_configuration.xml";
if(valide_fichier_configuration(fichier)==0)
  {
    if(mode_debug==1)
      printf("Attention, fichier configuration %s non valide\n",fichier);
    initialise_valeur_par_defaut(fichier);
  }
```

Note: Va sans doute arriver chez le client

=> tests avec if{...} (et pas avec assertions)

Utilisation de **exception**
(voir Java, C++, ...)

Glossaire des types d'erreurs

4. Erreur utilisateur (Ce n'est pas une *erreur)*

ex.
URL inexistante
Identifiant non connu dans la base
Demande d'un nombre positif, et recoit -12
...

Que faire:

Ne surtout pas quitter brutallement le programme (SegFault, abort) !
Message d'information à l'attention de l'utilisateur uniquement.
Doit être traité par une fonction spécifique de validation d'entrée utilisateur.

Note: Arrive forcément chez le client

=> tests avec if{...} (et pas avec assertions)

Gestion des erreurs

Types d'erreurs

→ **Erreur d'arrêt immédiats**

Transmission d'information d'erreur

- retour d'arguments
- passage de paramètres

Erreur utilisateur

Erreur d'arrêt immédiat

Approche manuelle

```c
#define AFFICHE_INFO_DEBUG printf("l.%d, %s, %s\n",__LINE__,__FUNCTION__,__FILE__)

//Contrat: recoit entier a de valeur absolue plus petite que 1000
void ma_fonction(int a)
{
  int contrat_valide=fabs(a<1000);
  if(contrat_valide==0)
    {
      //affiche le type d'erreur
      printf("Erreur a [%d] doit etre plus petit que 1000\n",a);
      //affiche les informations de localisation pour le debug
      AFFICHE_INFO_DEBUG;
      //quitte brutallement
      exit(1);
    }
}
```

renvoi 1 à la ligne de commande

ex. ma_fonction(10000) => Erreur a [10000] doit etre plus petit que 1000
l.13, ma_fonction, mon_fichier.c

Erreur d'arrêt immédiat

Approche par assertion


```c
#include <assert.h>

//Contrat: recoit entier a de valeur absolue plus petite que 1000
void ma_fonction(int a)
{
  assert(a<1000);

}
```

ex. ma_fonction(10000) =>

mon_executable: mon_fichier.c:8: ma_fonction: Assertion `a<1000' failed.
Aborted

Erreur d'arrêt immédiat

Comparaison if/assert

```
if(certificat==0)
{
  printf(<type d'erreur>+<info_debug>);
  exit(<valeur_erreur>) / abort();
}
```

```
assert(certificat);
```

+ de liberté:
  - message erreur
  - la manière de quitter

+ cours
  => **+lisible**
message erreur complet pour debug
  => **pas de risques d'oublis**
standard
  => **auto-documentation**

Conclusion 1: Pour des erreurs de **developpement**
=> Utilisez **assert**!

Erreur d'arrêt immédiat

Etude du fonctionnement de assert:

```c
#include <assert.h>

//Contrat: recoit entier a de valeur absolue plus petite que 1000
void ma_fonction(int a)
{
  assert(a<1000);

}
```

compilation mode debug

gcc -g mon_fichier.c -E -Wall -Wextra

```c
void ma_fonction(int a)
{
  ((a<1000) ? (void) (0) : __assert_fail ("a<1000", "mon_fichier.c", 7, __PRETTY_FUNCTION__));
}
```

-> utilisation des macros à votre place

Erreur d'arrêt immédiat

Etude du fonctionnement de assert:

```c
#include <assert.h>

//Contrat: recoit entier a de valeur absolue plus petite que 1000
void ma_fonction(int a)
{
  assert(a<1000);

}
```

compilation mode release

```
gcc -O2 mon_fichier.c -E
    -Wall -Wextra
    -DNDEBUG
```

```c
void ma_fonction(int a)
{
  ((void) (0));
}
```

-> condition non vérifiée en mode *release*

Erreur d'arrêt immédiat

Etude du fonctionnement de assert: 

Synthèse: Assert() ne vérifie la condition qu'en mode **debug**
=> pendant le developpement

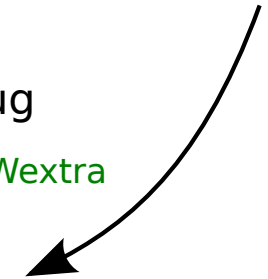
Avantage:	**Inconvénient:**
Pas de perte de temps en release	Ne peut être utilisé (seul) que pour des erreurs de developpement n'arrivant jamais sur la version finale

Conclusion: Assert() est une **aide pour le developpeur**!
Ce n'est pas une vérification constante

Erreur d'arrêt immédiat

Exemple de assert:

```c
//Contrat: recoit un pointeur non NULL
//Certification: la valeur du pointeur vaut 2
void ecrit_deux(int *pointeur)
{
  assert(pointeur!=NULL);
  *pointeur=2;
}

int main()
{
  int a;
  ecrit_deux(&a);
  ecrit_deux(NULL);
}
```



Mode debug

gcc -g -Wall -Wextra

Mode release

gcc -O2 -Wall -Wextra -DNDEBUG

```
mon_executable: erreur_01.c:8: ecrit_deux: Assertion `pointeur!=((void *)0)' failed.
Aborted
```

Segmentation fault

Erreur d'arrêt immédiat

Exemple de assert:

```
void demande_valeur_utilisateur()
{
  int nombre=0;

  printf("Donnez nombre entre 0 et 10.\n> ");
  scanf("%d",&nombre);

  assert(nombre>=0 && nombre<=10);

  printf("Vous avez choisi le nombre %d\n",nombre);
}
```

A ne pas faire!

X

=> Ne pas vérifier une entrée utilisateur avec un assert

-> quitte brutallement en mode debug = mauvais
-> ne vérifie rien en mode release = mauvais

Erreur d'arrêt immédiat

Manipulation de tableau sans dépassement mémoire:

```c
#define MAX_INDICE 4
struct vecteur
{
  int tableau[MAX_INDICE];
};

//Pre-requis:
//  Pointeur constant vers un vecteur. Pointeur non NULL.
//  Indice pointant dans ce vecteur: entier non signe < MAX_INDICE
//Certifie:
//  Renvoie la valeur du vecteur pointe par cet indice
int vecteur_get(const struct vecteur* v,unsigned int indice)
{
  //certification de developpement
  assert(v!=NULL);
  assert(indice<MAX_INDICE);

  return v->tableau[indice];
}

int main()
{
  struct vecteur v={ {1,2,4,5} };

  int a=vecteur_get(&v,2);
  int b=vecteur_get(&v,5);

  return 0;
}
```

Avantages:

Lors du développement:
Assure le non dépassement mémoire

a.out: erreur_01.c:20: vecteur_get: Assertion `indice<4' failed.

Lors de l'utilisation:
Aucune perte de performance par rapport à la version sans verification.

Erreur d'arrêt immédiat

Cumule aide au developpement / bugs a l'utilisation

```
//Pre-requis:
//  Pointeur constant vers un vecteur. Pointeur non NULL.
//  Indice pointant dans ce vecteur: entier non signe < MAX_INDICE
//Certifie:
//  Renvoie la valeur du vecteur pointe par cet indice
int vecteur_get(const struct vecteur* v,unsigned int indice)
{
  //certification de developpement
  assert(v!=NULL);
  assert(indice<MAX_INDICE);
  //mauvaise utilisation en production
  if(indice>=MAX_INDICE)
    {
      printf("Attention evenement inattendu survenue\n");
      printf("Contactez le developpeur avec les informations du fichier log.txt\n");

      //ecriture des informations de debugs dans log.txt
      FILE *fid=NULL;
      fid=fopen("log.txt","w");

      if(fid==NULL)//si on cumule les problemes
        {printf("Erreur ecriture fichier de log.txt, je quitte\n");abort();}

      fprintf(fid,"Erreur depassement indice %d>%d\n",indice,MAX_INDICE);
      fprintf(fid,"l.%d; %s; %s\n",__LINE__,__FUNCTION__,__FILE__);

      fclose(fid);
      return -1;//retourne une valeur par defaut => evite la seg-fault
    }

  return v->tableau[indice];
}
```

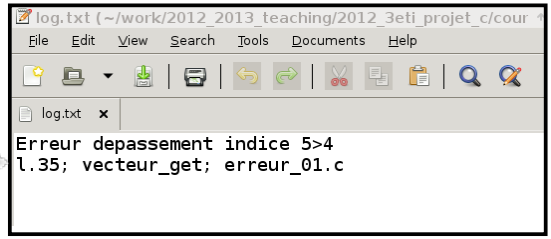
gcc -O2 -Wall -Wextra -DNDEBUG

```
Attention evenement inattendu survenue
Contactez le developpeur avec les informations du fichier log.txt
```

\+ le programme ne s'arrête pas

\+

log.txt

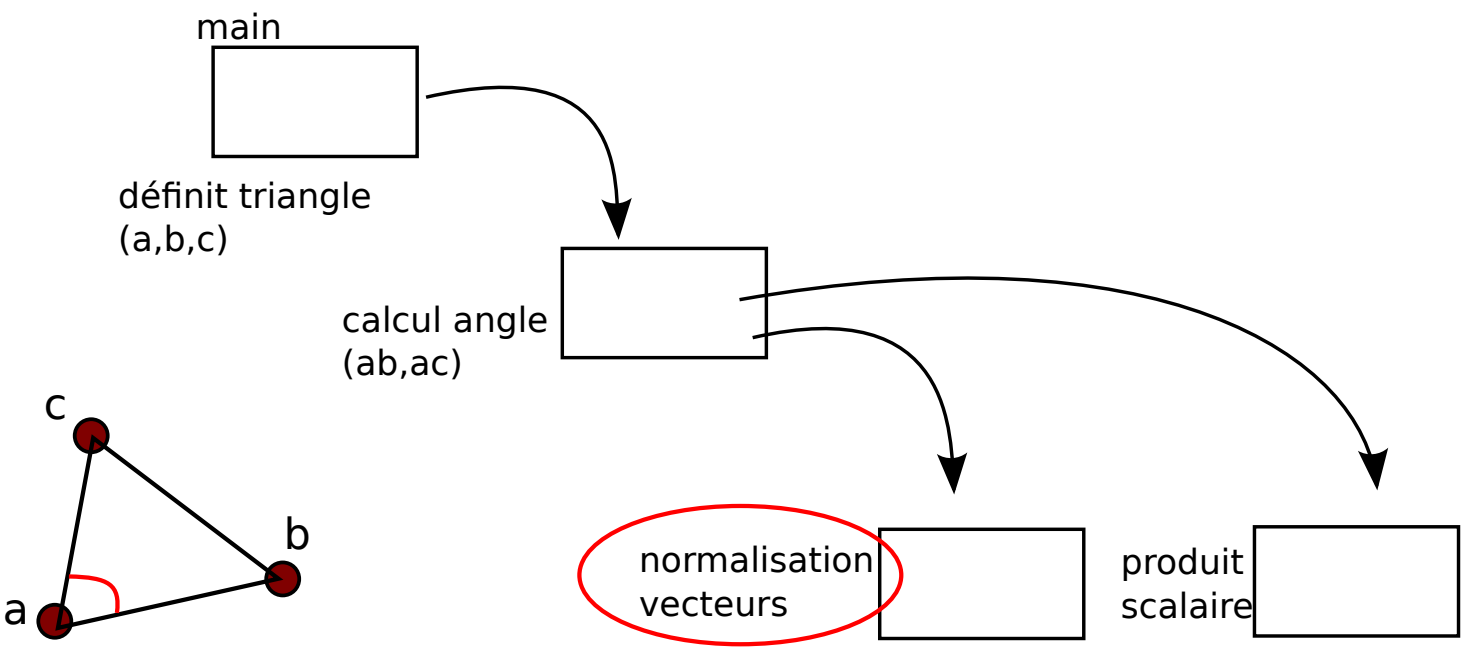


Gestion des erreurs

Types d'erreurs
Erreur d'arrêt immédiats
→ **Transmission d'information d'erreur**
- retour d'arguments
- passage de paramètres
Erreur utilisateur

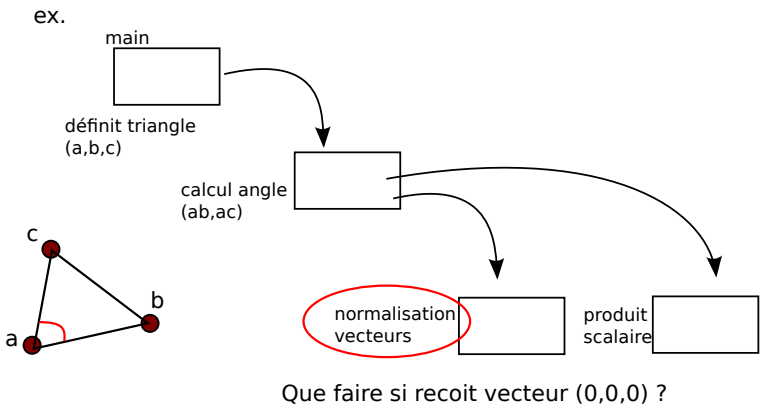
Erreur gérée par la fonction appelante

ex.



Que faire si recoit vecteur (0,0,0) ?

Erreur gérée par la fonction appelante



Dans normalisation vecteurs (localement), on ne peut que:

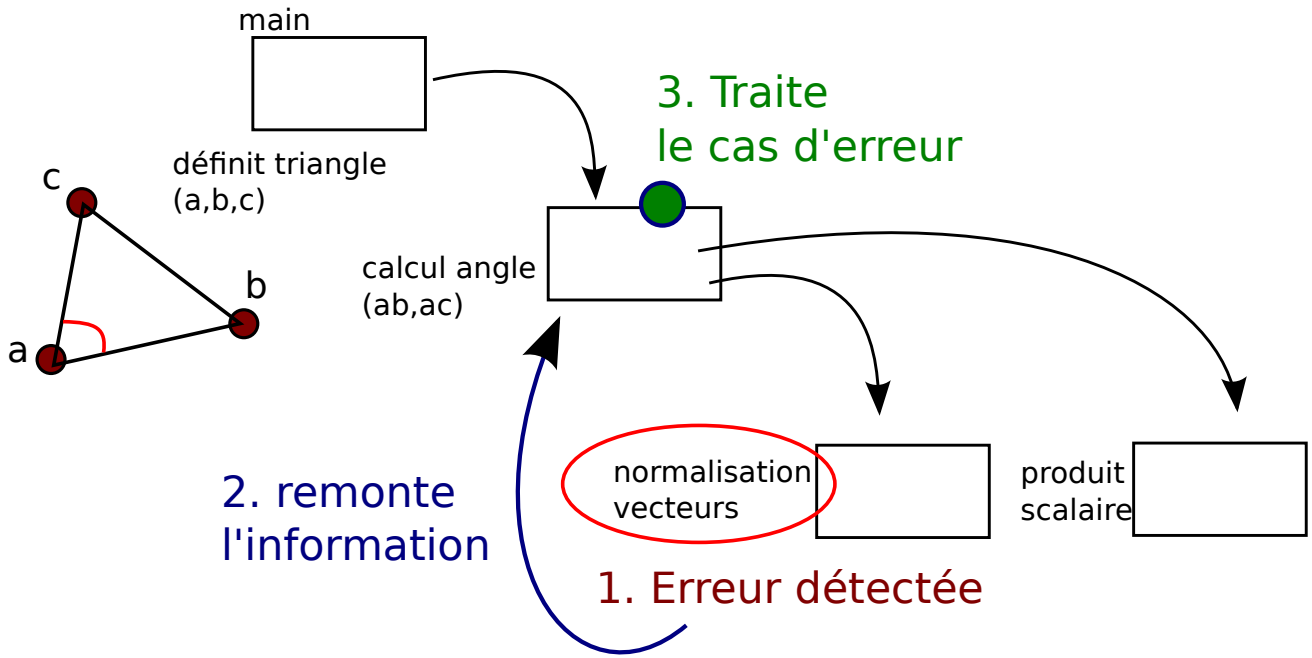
* Quitter => acceptable ?
* Renvoyer une valeur par défaut => laquelle?

Mieux:
Indiquer à *calcul_angle*, ou dans le *main* le problème

renvoi un angle de 0 par exemple

Erreur gérée par la fonction appelante

Il faut faire remonter l'information d'une erreur



Erreur gérée par la fonction appelante

Plusieurs solutions possible:

* Valeur de retour
* Variable globales
* Passage d'arguments par pointeurs

un mix de tous

Hors C
* Exception

But:

- Garder un programme lisible!
- Garder un programme maintenable
- Rester transparent en cas de bon fonctionnement
  (sinon perte de lisiblité)
- Pas trop transparent
  (sinon oublie de gestion des cas d'erreurs)

Erreur par retour d'arguments

fonction:

```c
#define FAIL 0
#define OK   1

int ecrit_fichier(const char *filename)
{
  assert(filename!=NULL);

  FILE *fid=NULL;
  fid=fopen(filename,"w");

  if(fid==NULL)
    return FAIL; //erreur

  int nbr_lettre=fprintf(fid,"Coucou c'est moi\n");
  if(nbr_lettre!=18)
    return FAIL; //erreur

  int ret=fclose(fid);
  if(ret!=0)
    return FAIL; //erreur

  return OK;
}
```

Avantage:

- utilisation assez élégante
  => if(ma_fonction())
- potentiellement transparent
  => possibilité de non vérification

Inconvénient:

- Empêche le retour d'arguments

utilisation:

```c
int main()
{
  if(ecrit_fichier("mon_fichier.txt")==FAIL)
    {
      printf("mon_fichier.txt n'est pas accessible, tente d'ecrire sur mon_fichier_2.txt\n");

      if(ecrit_fichier("mon_fichier_2.txt")==FAIL)
        {
          printf("Ce n'est pas mon jour de chance, je fais autre chose");
          faire_autre_chose();
        }

    }

  return 0;
}
```

Erreur par retour d'arguments

```c
#define FAIL 0
#define OK 1


struct vecteur
{float x,y,z;};

//Prend en parametre un vecteur et retourne le vecteur unitaire de meme direction
int vecteur_unitaire(struct vecteur vec,struct vecteur* resultat)
{
  assert(resultat!=NULL);

  float norme=sqrt(vec.x*vec.x+vec.y*vec.y+vec.z*vec.z);

  if(norme<1e-5)
    return FAIL;

  resultat->x = vec.x/norme;
  resultat->y = vec.y/norme;
  resultat->z = vec.z/norme;

  return OK;

}
```

```c
int main()
{
  struct vecteur a={1,2,3};struct vecteur ua={1,1,1};
  struct vecteur b={0,0,0};struct vecteur ub={1,1,1};

  int ret_1=vecteur_unitaire(a,&ua);
  int ret_2=vecteur_unitaire(b,&ub);

  if(ret_1!=OK || ret_2!=OK)
    {
      printf("Attention, vecteur de norme nulle\n");
      printf("Angle = 0.0\n");
      return 0;
    }

  printf("Angle = %f\n",ua.x*ub.x+ua.y*ub.y+ua.z*ub.z);

  return 0;
}
```

peu lisible, pas pratique.

on préfèrerait:
struct vecteur ua=vecteur_unitaire(a);

Erreur par retour d'arguments

```c
struct vecteur
{float x,y,z;};

float vecteur_norme(struct vecteur vec)
{
  return sqrt(vec.x*vec.x+vec.y*vec.y+vec.z*vec.z);
}

int vecteur_est_norme_non_nulle(struct vecteur vec)
{
  float epsilon=1e-5;
  return vecteur_norme(vec)>epsilon;
}

//Prend en parametre un vecteur et retourne le vecteur unitaire de meme direction
// Prerequis: Un vecteur de norme non nulle
// Certifie: Un vecteur de meme direction et de norme unitaire
struct vecteur vecteur_unitaire(struct vecteur vec)
{
  assert(vecteur_est_norme_non_nulle(vec)==VRAI);

  float norme=vecteur_norme(vec);
  struct vecteur resultat={vec.x/norme,vec.y/norme,vec.z/norme};

  return resultat;
}
```

fonction "binaire" de vérification amont

utilisation de contrat pour éviter les erreurs de codage

```c
int main()
{
  struct vecteur a={1,2,3};struct vecteur ua={1,1,1};
  struct vecteur b={0,0,0};struct vecteur ub={1,1,1};

  if(vecteur_est_norme_non_nulle(a)==FAUX || vecteur_est_norme_non_nulle(b)==FAUX)
    {
      printf("Attention, vecteur de norme nulle\n");
      printf("Angle = 0.0\n");
      return 0;
    }

  ua=vecteur_unitaire(a);
  ub=vecteur_unitaire(b);

  printf("Angle = %f\n",ua.x*ub.x+ua.y*ub.y+ua.z*ub.z);

  return 0;
}
```

traitement spécifique erreur "utilisateur"

+ lisible
+ logique

Erreur par retour d'arguments

Le type d'erreur peut être mixé avec le résultat **A éviter!**

```c
#define TAILLE_TABLEAU 6

int recupere_note(const int* tableau_note,int indice)
{
  if(tableau_note==NULL)
    return -1;
  if(indice<0)
    return -2;
  if(indice>TAILLE_TABLEAU)
    return -3;

  return tableau_note[indice];
}

int main()
{
  //les notes sont comprises entre 0 et 20
  int tableau_note[TAILLE_TABLEAU]={12,14,5,14,16,18};

  int valeur_erreur=recupere_note(tableau_note,12);
  if(valeur_erreur<0)
    {
      switch(valeur_erreur)
        {
        case -1:
          printf("pointeur NULL\n");break;
        case -2:
          printf("indice negatif\n");break;
        case -3:
          printf("indice trop grand\n");break;
        }
    }
  else
    printf("Note %d\n",valeur_erreur);
}
```

Avantage:

Compact (si peu de mémoire)

Effet de bords

(si le domaine de validité des données changent)

Inconvénient:

Perte de lisibilité

Mélange données/erreurs

Documentation difficile

=> A éviter

Erreur par retour d'arguments

mélange code erreur / valeurs par mask de bits


```
//couleur:
//0xEERRVVBB
//EE: erreur -> 01: indice trop petit
//    erreur -> 02: indice trop grand
//RR: rouge
//VV: vert
//BB: bleu

#define TAILLE_TABLEAU 4

int recupere_couleur(int tableau_couleur[],int indice)
{
  if(indice<0)
    return 0x01000000;
  if(indice>TAILLE_TABLEAU)
    return 0x02000000;

  return tableau_couleur[indice];
}
```

Concaténation (code_erreur,rouge,vert,bleu)

A NE PAS FAIRE!

Inconvénients:

- difficile à lire
- documentation indispensable
- pas évolutif
  (ajout canal alpha impossible)

```
int main()
{
  //les notes sont comprises entre 0 et 20
  int tableau_couleur[TAILLE_TABLEAU]={0x000000,0xFF0000,0x00FF00,0x0000FF};

  int valeur_erreur=recupere_couleur(tableau_couleur,8);
  if(valeur_erreur>>24)
    {
      switch(valeur_erreur>>24)
        {
        case 1:
          printf("indice negatif\n");break;
        case 2:
          printf("indice trop grand\n");break;
        }
    }
  else
    printf("Couleur (%x,%x,%x)\n",
           valeur_erreur>>16,
           (valeur_erreur&0x00FFFF)>>8,
           (valeur_erreur&0x0000FF));
}
```

X

peu lisible

=> Ne pas utiliser ce genre d'approche pour du developpement d'un logiciel.

OK uniquement pour embarqué limité

Erreur par passage de paramètre

Cas d'erreur écrit dans un parametre passé par pointeur
(si pointeur NULL => pas de prise en compte)

```c
#define FAUX 0
#define VRAI 1

struct vecteur
{float x,y,z;};

float vecteur_norme(struct vecteur vec)
{
  return sqrt(vec.x*vec.x+vec.y*vec.y+vec.z*vec.z);
}

struct vecteur vecteur_unitaire(struct vecteur vec,int *execution_ok)
{
  struct vecteur resultat={1,0,0};

  float norme=vecteur_norme(vec);
  if(norme<1e-5)
    {
      if(execution_ok!=NULL)
        *execution_ok=FAUX;
      return resultat;
    }
  resultat.x=vec.x/norme;
  resultat.y=vec.y/norme;
  resultat.z=vec.z/norme;

  if(execution_ok!=NULL)
    *execution_ok=VRAI;

  return resultat;
}
```

prise en compte de l'erreur

```c
int main()
{
  struct vecteur a={0,0,0};

  int execution_ok=VRAI;
  struct vecteur ua=vecteur_unitaire(a,&execution_ok);

  if(execution_ok==FAUX)
    {
      printf("Erreur de normalisation de vecteur\n");
      return 0;
    }

  struct vecteur ub=vecteur_unitaire(a,NULL);


  return 0;
}
```

on ne tiens pas compte du cas d'erreur

Erreur par passage de paramètre

Cas d'erreur écrit dans un parametre passé par pointeur
(si pointeur NULL => pas de prise en compte)

```c
#define FAUX 0
#define VRAI 1

struct vecteur
{float x,y,z;};

float vecteur_norme(struct vecteur vec)
{
  return sqrt(vec.x*vec.x+vec.y*vec.y+vec.z*vec.z);
}

struct vecteur vecteur_unitaire(struct vecteur vec,int *execution_ok)
{
  struct vecteur resultat={1,0,0};

  float norme=vecteur_norme(vec);
  if(norme<1e-5)
    {
      if(execution_ok!=NULL)
        *execution_ok=FAUX;
      return resultat;
    }
  resultat.x=vec.x/norme;
  resultat.y=vec.y/norme;
  resultat.z=vec.z/norme;

  if(execution_ok!=NULL)
    *execution_ok=VRAI;

  return resultat;
}
```

prise en compte de l'erreur

```c
int main()
{
  struct vecteur a={0,0,0};

  int execution_ok=VRAI;
  struct vecteur ua=vecteur_unitaire(a,&execution_ok);

  if(execution_ok==FAUX)
    {
      printf("Erreur de normalisation de vecteur\n");
      return 0;
    }

  struct vecteur ub=vecteur_unitaire(a,NULL);


  return 0;
}
```

on ne tiens pas compte du cas d'erreur

Avantage:

* permet l'écriture a=fonction(b,...)
=> utilisation plus lisible

* possibilitée de ne pas tenir compte de l'erreur
mais reste visible

Inconvénient:

* documentation nécessaire
(parametre NULL?)

* syntaxe fonction moins lisible

Erreur par passage de paramètre

Possibilité de travailler avec une struct
(extensibilité, lisibilité)

```c
//les differents types d'erreur possibles
enum type_erreur {pointeur_null,indice_trop_grand,indice_trop_petit};

#define TAILLE_MAX 256
struct code_erreur
{
  int actif; //est ce qu'une erreur existe

  enum type_erreur type; //le type d'erreur
  int ligne; //la ligne d'ou provient l'erreur
  char nom_fonction[TAILLE_MAX]; //le nom de la fonction
                                 //d'ou provient l'erreur
};
```

structure d'erreur
+ contient toutes les infos de debug
+ lisible
=> extensible

```c
int main()
{
  int tableau[TAILLE_TABLEAU]={4,2,-1,4};

  int a=recupere_valeur(tableau,2,NULL);

  struct code_erreur erreur={ FAUX,0,0,"\0" };
  int b=recupere_valeur(tableau,-3,&erreur);

  if(erreur.actif==VRAI)
    traitement_erreur(&erreur);

  return 0;
}
```

utilisation
transparente

gestion du cas d'erreur

Erreur par passage de paramètre

Possibilité de travailler avec une struct (extensibilité, lisibilité)

```c
//les differents types d'erreur possibles
enum type_erreur {pointeur_null,indice_trop_grand,indice_trop_petit};

#define TAILLE_MAX 256
struct code_erreur
{
  int actif; //est ce qu'une erreur existe

  enum type_erreur type; //le type d'erreur
  int ligne; //la ligne d'ou provient l'erreur
  char nom_fonction[TAILLE_MAX]; //le nom de la fonction
                                 //d'ou provient l'erreur
};
```

```c
#define VRAI 1
#define FAUX 0

#define TAILLE_TABLEAU 4

int recupere_valeur(const int* tableau,int indice,struct code_err
eur *retour_erreur)
{
  if(tableau==NULL)
    {
      if(retour_erreur!=NULL)
        {
          retour_erreur->actif=VRAI;
          retour_erreur->type=pointeur_null;
          retour_erreur->ligne=__LINE__;
          strcpy(retour_erreur->nom_fonction,__FUNCTION__);
        }
      return -1;
    }

  if(indice<0)
    {
      if(retour_erreur!=NULL)
        {
          retour_erreur->actif=VRAI;
          retour_erreur->type=indice_trop_petit;
          retour_erreur->ligne=__LINE__;
          strcpy(retour_erreur->nom_fonction,__FUNCTION__);
        }
      return -1;
    }

  if(indice>=TAILLE_TABLEAU)
    {
      if(retour_erreur!=NULL)
        {
          retour_erreur->actif=VRAI;
          retour_erreur->type=indice_trop_grand;
          retour_erreur->ligne=__LINE__;
          strcpy(retour_erreur->nom_fonction,__FUNCTION__);
        }
      return -1;
    }

  if(retour_erreur!=NULL)
    retour_erreur->actif=FAUX;

  return tableau[indice];
}
```

une implémentation possible
de la fonction, et du traitement de l'erreur:

```c
void traitement_erreur(const struct code_erreur* e)
{
  assert(e->actif==VRAI);

  printf("Erreur detecte\n");
  printf("Erreur de type %d\n",e->type);
  printf("a la ligne %d, dans la fonction %s\n",
         e->ligne,e->nom_fonction);
}
```

Erreur par passage de paramètre

Communication par variable globale

```c
//constantes de gestion d'erreurs
#define PAS_ERREUR 0
#define ERREUR_DIVISION_ZERO 1

//variable globale d'erreur
int variable_globale_erreur=PAS_ERREUR;


struct vecteur
{float x,y,z;};

struct vecteur vecteur_unitaire(struct vecteur vec)
{
  float norme=vec.x*vec.x+vec.y*vec.y+vec.z*vec.z;
  float epsilon=1e-5;

  struct vecteur resultat={-1,-1,-1};
  if(norme<epsilon)
    {
      variable_globale_erreur=ERREUR_DIVISION_ZERO;
      return resultat;
    }

    resultat.x=vec.x/norme;
    resultat.y=vec.y/norme;
    resultat.z=vec.z/norme;

    return resultat;
}
```

```c
int main()
{
  struct vecteur a={1,2,3};
  struct vecteur b={0,0,0};

  struct vecteur ua=vecteur_unitaire(a);
  struct vecteur ub=vecteur_unitaire(b);

  if(variable_globale_erreur!=PAS_ERREUR)
    {printf("Attention, vecteur de norme nulle detectee \n");return 1;}

  return 0;
}
```

Avantage:
- transparent
- approche standard C

Inconvénient:
- trop transparent
  => absence de vérification
- documentation nécessaire

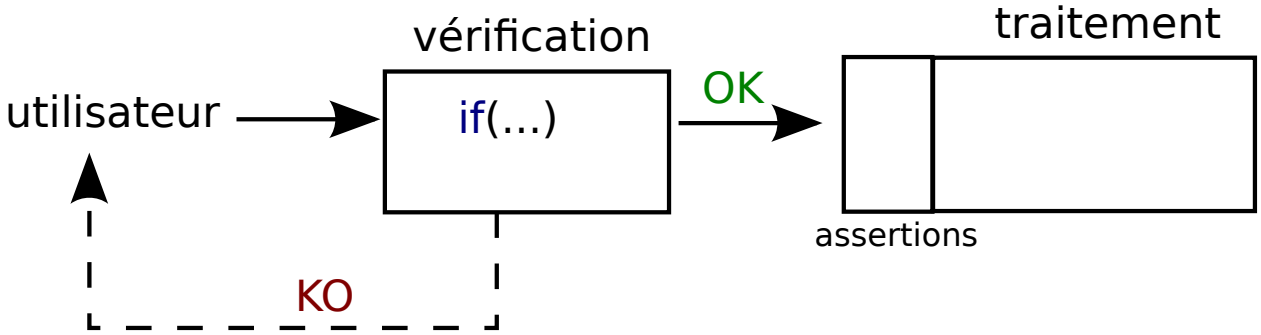
Gestion des erreurs

Types d'erreurs
Erreur d'arrêt immédiats
Transmission d'information d'erreur
- retour d'arguments
- passage de paramètres

→ **Erreur utilisateur**

Erreur utilisateur

Séparer vérification du traitement



But:

Construire des blocs unitaires (saisie, traitement)
Chaque bloc est certifié
Chaque bloc possède la gestion d'erreur appropriée

Erreur utilisateur

ex.
Programme affichant un nom de fichier existant donné par l'utilisateur

A EVITER!

```c
#define TAILLE_MAX 128
#define TAILLE_LIGNE 256

int main()
{
    char filename[TAILLE_MAX];

    int fichier_ok=0;
    do
    {
        //saisie
        printf("Indiquez chemin vers fichier:\n> ");
        scanf("%128s",filename);

        //test existance
        FILE *fid=NULL;
        fid=fopen(filename,"r");

        if(fid!=NULL)
        {
            fichier_ok=1;

            //lecture et affichage du fichier
            char buffer[TAILLE_LIGNE];
            while(fgets(buffer,TAILLE_LIGNE,fid) != NULL)
                printf("%s",buffer);
            fclose(fid);
            fid=NULL;
        }
        else
            printf("Fichier %s n'existe pas\n\n",filename);
    }while(fichier_ok!=1);

    return 0;
}
```

partie utilisateur

partie traitement

partie utilisateur

Désavantage:

Mélange utilisateur/traitement
=> difficile à comprendre/lire

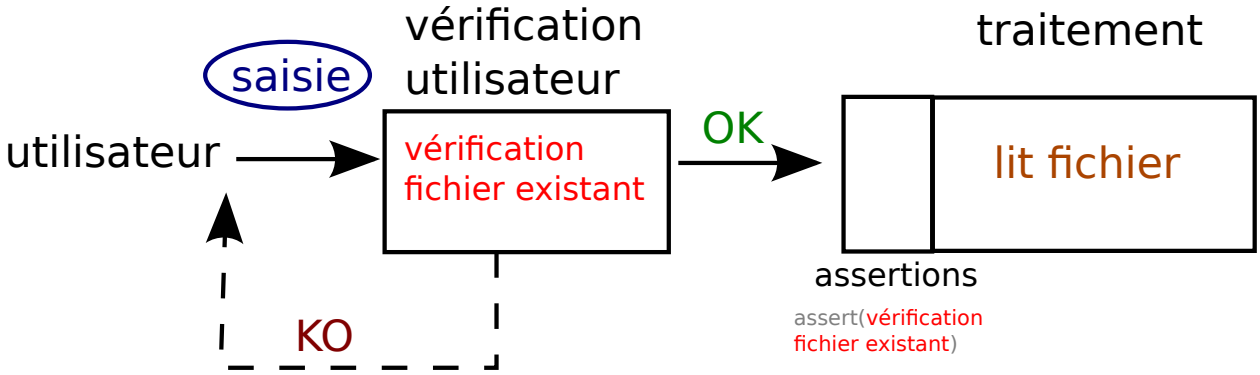
traitement non unitaire
=> difficile à debugger/tester

Traitement et saisie non réutilisable dans un autre contexte

Erreur utilisateur

ex.
Programme affichant un nom de fichier existant donné par l'utilisateur

Amélioration possible:

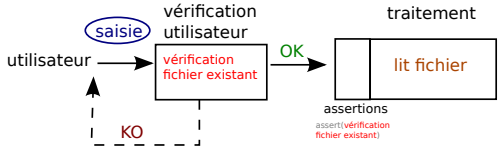


saisie = demander nom de fichier en ligne de commande
vérification fichier existant = ouvrir fichier, vérifier que l'ouverture est correcte
lit fichier = ouvre fichier, lit ligne après ligne jusqu'à fin, ferme fichier.

Erreur utilisateur

ex.
Programme affichant un nom de fichier existant donné par l'utilisateur

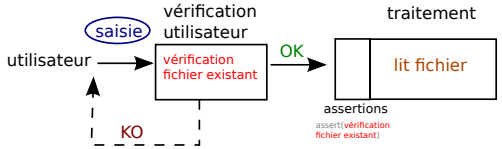
```c
//Donnez le nom d'un fichier a lire

#define TAILLE_MAX 128
#define TAILLE_LIGNE 256

//Saisie utilisateur d'un fichier
//Demande a l'utilisateur de saisir le chemin vers
// un fichier existant.
void saisie_fichier_existant(char filename[]);

//Verifie si un fichier est accessible en mode lecture
//Prerequis:
//     Un nom de fichier sous forme de chaine de caracteres.
//Certifie:
//     Retourne 0 si le fichier n'est pas accessible en mode lecture
//     Retourne 1 si le fichier est accessible en mode lecture
int est_fichier_existant(const char filename[]);

//Lecture d'un fichier existant
//Prerequis:
//     Un nom de fichier qui existe deja.
//Certifie:
//     L'affichage sur la ligne de commande du fichier.
void lecture_fichier(char filename[]);
```




```c
int main()
{
  char filename[TAILLE_MAX]="\0";

  saisie_fichier_existant(filename); //bloc saisie utilisateur
  lecture_fichier(filename);         //bloc traitement

  return 0;
}
```

Erreur utilisateur

ex.
Programme affichant un nom de fichier existant donné par l'utilisateur



Implémentation possible:

```c
int est_fichier_existant(const char filename[])
{
  FILE *fid=NULL;
  fid=fopen(filename,"r");
  if(fid==NULL)
    return 0;

  int ret=fclose(fid);
  if(ret!=0){printf("Probleme fermeture fichier %s\n",filename);}
  fid=NULL;
  return 1;
}
```

```c
void lecture_fichier(char filename[])
{
  assert(est_fichier_existant(filename));

  FILE *fid=NULL;
  fid=fopen(filename,"r");

  assert(fid!=NULL);

  //lecture et affichage du fichier
  char buffer[TAILLE_LIGNE];
  while(fgets(buffer,TAILLE_LIGNE,fid) != NULL)
    printf("%s",buffer);

  int ret=fclose(fid);
  assert(ret==0);
  fid=NULL;

}
```

```c
void saisie_fichier_existant(char filename[])
{
  int fichier_ok=0;

  do
    {
      //saisie
      printf("Indiquez chemin vers fichier:\n> ");
      scanf("%128s",filename);

      //test existance
      fichier_ok=est_fichier_existant(filename);

      if(fichier_ok!=1)
        printf("Fichier %s n'existe pas\n\n",filename);

    }while(fichier_ok!=1);
}
```

Méthode de debug

Méthode manuelle (printf)
Debugger à points d'arrêts
Détecteur fuites mémoires

Methode de debug

Bugs de:

Fonctionnement globale
=> Principe de tests: unitaire + intégrations

Erreurs mémoires
=> utilisation de debuggers

Methode de debug

Segmentation fault
Erreur de segmentation

= erreur mémoire

= écriture/accès à une zone mémoire
non autorisée

(C'est le noyaux qui lance une segfault: pas le programme!)

90% du temps:
Segmentation fault due à :

* depassement de tableau (indice trop grand ou <0)
* écriture sur pointeur non alloué
* désallocation sur une adresse invalide (free)

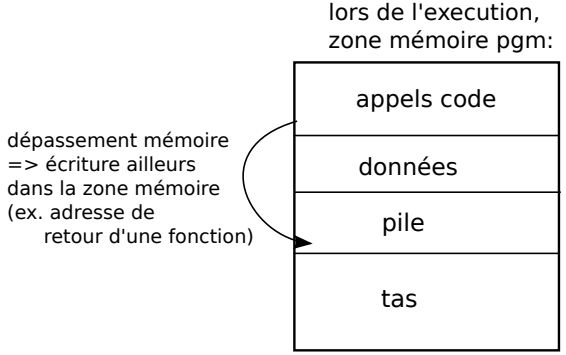
Methode de debug

Difficile à débugger car:

* Segfault ou comportement inattendu **non détecté** au moment de l'erreur
* Une erreur mémoire n'aboutit **pas forcément** à une segmentation fault

    * peut aboutir à un comportement indeterminé (aléatoire en fonction des executions)
    * peut ne pas apparaitre
    * peut modifier le comportement hors de la logique du C
    * le comportement suspect n'apparait pas au moment de l'erreur, mais plus loin dans le programme.

ex typiques:
* *Valeur d'un tableau ou d'une variable modifiée*
* *Comportement du programme différent lors de l'ajout de commentaires dans le code*
* *Le return d'une fonction n'est pas récupéré par la fonction appelante*



Méthode de debug

→ **Méthode manuelle (printf)**
Debugger à points d'arrêts
Détecteur fuites mémoires

Debug par affichage: printf

```c
#define TAILLE_MAX 15

void derivation(int valeurs[],int derivee[])
{
    int k=0;
    for(k=0;k<TAILLE_MAX;++k)
        derivee[k] = valeurs[k+1]-valeurs[k];

}

int main()
{
    int valeurs[TAILLE_MAX];
    int derivee[TAILLE_MAX];

    int k=0;
    for(k=0;k<TAILLE_MAX;++k)
        valeurs[k]=3*k*k;

    derivation(valeurs,derivee);

    return 0;
}
```

```c
#define TAILLE_MAX 15

void derivation(int valeurs[],int derivee[])
{
    int k=0;
    for(k=0;k<TAILLE_MAX;++k)
    {
        printf("%d/%d -> %d\n",k,TAILLE_MAX,valeurs[k]);
        printf("%d/%d -> %d\n",k+1,TAILLE_MAX,valeurs[k+1]);
        derivee[k] = valeurs[k+1]-valeurs[k];
    }

}

int main()
{
    int valeurs[TAILLE_MAX];
    int derivee[TAILLE_MAX];

    int k=0;
    for(k=0;k<TAILLE_MAX;++k)
        valeurs[k]=3*k*k;

    derivation(valeurs,derivee);

    return 0;
}
```

```
0/15 -> 0
1/15 -> 3
1/15 -> 3
2/15 -> 12
2/15 -> 12
...
13/15 -> 507
13/15 -> 507
14/15 -> 588
14/15 -> 588
15/15 -> 15
```

Debug par affichage: printf

Comportement inattendu:

```c
int main()
{
    char mot_1[]="coucou";
    char mot_2[]="bonjour monsieur";

    strcpy(mot_2," en fin de journee, nous disons bonsoir");

    printf("%s\n",mot_1);

    return 0;
}
```

```
./mon_executable
> bonsoir
```

```c
int main()
{
    char mot_1[]="coucou";
    char mot_2[]="bonjour monsieur";

    printf("%d/%d\n",sizeof(mot_2)/sizeof(char),
           strlen(" en fin de journee, nous disons bonsoir")+1);

    strcpy(mot_2," en fin de journee, nous disons bonsoir");

    printf("%s\n",mot_1);

    return 0;
}
```

```
./mon_executable
> 17/40
> bonsoir
```

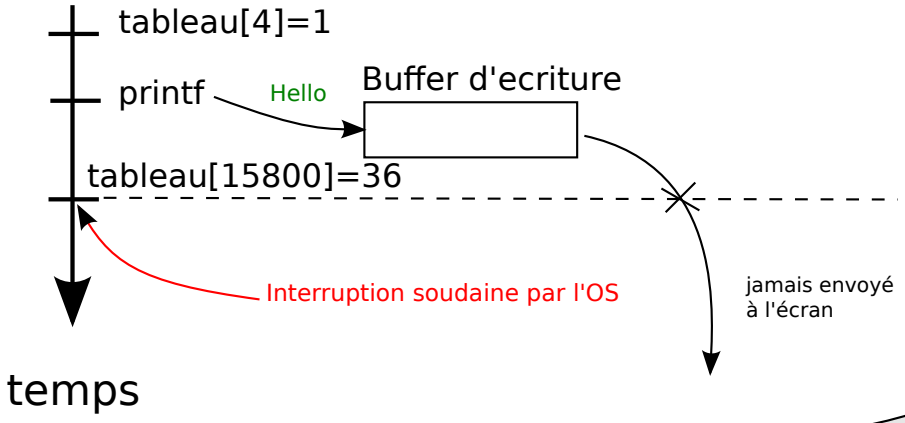
Debug par affichage: printf

Attention à l'effet du tampon d'écriture

```c
int main()
{
  int tableau[5];

  tableau[4]=1;

  printf("Hello");

  tableau[15800]=36;

  printf("World");

  return 0;
}
```

```
./mon_executable
> Segmentation fault
```

Pourtant seg-fault à la ligne suivante!!



Debug par affichage: printf

Attention à l'effet du tampon d'écriture

```c
int main()
{
  int tableau[5];

  tableau[4]=1;

  printf("Hello");

  tableau[15800]=36;

  printf("World");

  return 0;
}
```

./mon_executable
> Segmentation fault

Pourtant seg-fault à la ligne suivante!!

Solution: Toujours vider le buffer (flush)

- Retour à la ligne "\n"
  (écran uniquement)

- commande *fflush(stdout)*;

```c
int main()
{
  int tableau[5];

  tableau[4]=1;

  printf("Hello\n");

  tableau[15800]=36;

  printf("World\n");

  return 0;
}
```

déduit: seg-fault entre "Hello" et "World"

./mon_executable
> Hello
> Segmentation fault

Debug par affichage: printf

Avantages:

+ léger, rapide
  (pas d'outils externes)
+ rapide
+ execution et debug du vrai code
  (pas d'émulation)

Inconvénients:

- Nécessite de bien connaitre le code
- Nécessite de pré-localiser l'erreur
- Précaution lors de l'utilisation de données nombreuses

Synthèse:

Méthode rapide lorsque l'on travaille sur son propre code (vérification d'une boucle, d'un pointeur, etc.)

Peu adapté pour le débug d'un code externe.

Méthode de debug

Méthode manuelle (printf)

→ **Debugger à points d'arrêts**

Détecteur fuites mémoires

Debug par debugger : semi-automatique

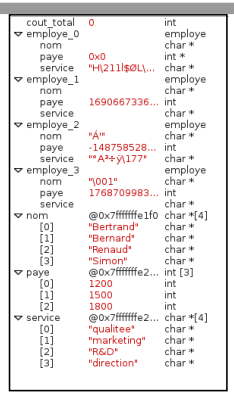
```c
#include <stdio.h>

struct employe
{
  char *nom;
  int *paye;
  char *service;
};

int main()
{

    char *nom[]={"Bertrand","Bernard","Renaud","Simon"};
    int paye[]={1200,1500,1800};
    char *service[]={"qualitee","marketing","R&D","direction"};

    struct employe employe_0={nom[2],&paye[0],service[1]};
    struct employe employe_1={nom[1],&paye[0],service[0]};
    struct employe employe_2={nom[0],&paye[2],service[2]};
    struct employe employe_3;employe_3.nom=nom[3];

    int cout_total=0;
    cout_total=*employe_0.paye+
            *employe_1.paye+
            *employe_2.paye+
            *employe_3.paye;


    return 0;

}
```

cout_total	0	int
employe_0		employe
nom		char *
paye	0x0	int *
service	"H\211l$ØL\...	char *
employe_1		employe
nom		char *
paye	1690667336...	int
service		char *
employe_2		employe
nom	"Á'"	char *
paye	-148758528...	int
service	"°A³÷ÿ\177"	char *
employe_3		employe
nom	"\001"	char *
paye	1768709983...	int
service		char *
nom	@0x7fffffffe1f0	char *[4]
[0]	"Bertrand"	char *
[1]	"Bernard"	char *
[2]	"Renaud"	char *
[3]	"Simon"	char *
paye	@0x7fffffffe2...	int [3]
[0]	1200	int
[1]	1500	int
[2]	1800	int
service	@0x7fffffffe2...	char *[4]
[0]	"qualitee"	char *
[1]	"marketing"	char *
[2]	"R&D"	char *
[3]	"direction"	char *

point d'arrêt

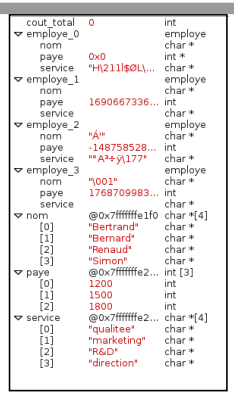
variables + valeurs

Debug par debugger : semi-automatique

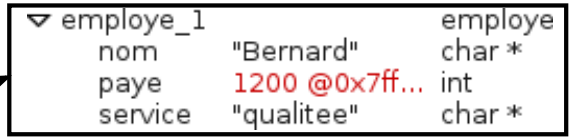



```c
#include <stdio.h>

struct employe
{
  char *nom;
  int *paye;
  char *service;
};

int main()
{

    char *nom[]={"Bertrand","Bernard","Renaud","Simon"};
    int paye[]={1200,1500,1800};
    char *service[]={"qualitee","marketing","R&D","direction"};

    struct employe employe_0={nom[2],&paye[0],service[1]};
    struct employe employe_1={nom[1],&paye[0],service[0]};
    struct employe employe_2={nom[0],&paye[2],service[2]};
    struct employe employe_3;employe_3.nom=nom[3];

    int cout_total=0;
    cout_total=*employe_0.paye+
            *employe_1.paye+
            *employe_2.paye+
            *employe_3.paye;


    return 0;

}
```

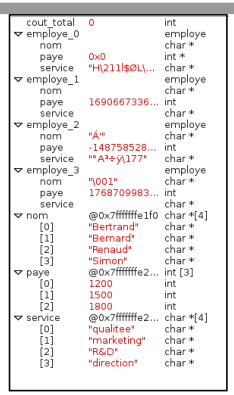
Debug par debugger : semi-automatique




```c
#include <stdio.h>

struct employe
{
  char *nom;
  int *paye;
  char *service;
};

int main()
{

    char *nom[]={"Bertrand","Bernard","Renaud","Simon"};
    int paye[]={1200,1500,1800};
    char *service[]={"qualitee","marketing","R&D","direction"};

    struct employe employe_0={nom[2],&paye[0],service[1]};
    struct employe employe_1={nom[1],&paye[0],service[0]};
    struct employe employe_2={nom[0],&paye[2],service[2]};
    struct employe employe_3;employe_3.nom=nom[3];

    int cout_total=0;
    cout_total=*employe_0.paye+
            *employe_1.paye+
            *employe_2.paye+
            *employe_3.paye;


    return 0;

}
```

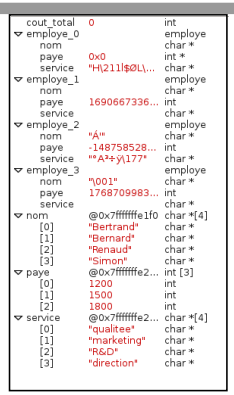


Debug par debugger : semi-automatique

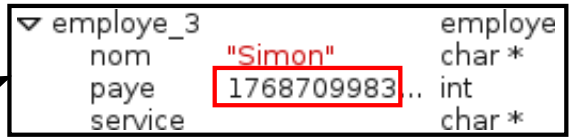



```c
#include <stdio.h>

struct employe
{
  char *nom;
  int *paye;
  char *service;
};

int main()
{

    char *nom[]={"Bertrand","Bernard","Renaud","Simon"};
    int paye[]={1200,1500,1800};
    char *service[]={"qualitee","marketing","R&D","direction"};

    struct employe employe_0={nom[2],&paye[0],service[1]};
    struct employe employe_1={nom[1],&paye[0],service[0]};
    struct employe employe_2={nom[0],&paye[2],service[2]};
    struct employe employe_3;employe_3.nom=nom[3];

    int cout_total=0;
    cout_total=*employe_0.paye+
            *employe_1.paye+
            *employe_2.paye+
            *employe_3.paye;


    return 0;

}
```

Debug par debugger : semi-automatique




```c
#include <stdio.h>

struct employe
{
  char *nom;
  int *paye;
  char *service;
};

int main()
{

    char *nom[]={"Bertrand","Bernard","Renaud","Simon"};
    int paye[]={1200,1500,1800};
    char *service[]={"qualitee","marketing","R&D","direction"};

    struct employe employe_0={nom[2],&paye[0],service[1]};
    struct employe employe_1={nom[1],&paye[0],service[0]};
    struct employe employe_2={nom[0],&paye[2],service[2]};
    struct employe employe_3;employe_3.nom=nom[3];

    int cout_total=0;
    cout_total=*employe_0.paye+
            *employe_1.paye+
            *employe_2.paye+
            *employe_3.paye;


    return 0;

}
```



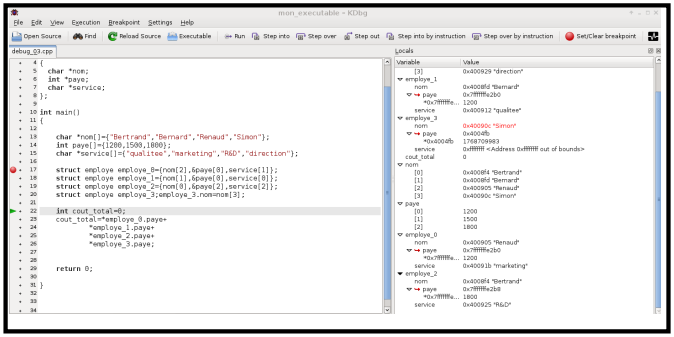
Debug par debugger : semi-automatique

Outils de debugs par points d'arrets:

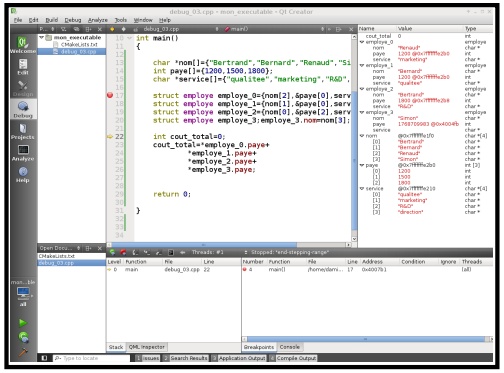
gdb: mode texte

cs.brynmawr.edu/cs312/gdb-tutorial-handout.pdf

kdbg:



QtCreator:



Debug par debugger : semi-automatique

Avantage:

+ Visuelle, vision globale
+ debug données complexes
(listes chainées, graphes, ...)

Inconvénient:

- Localisation préalable
(fichier,lignes,variables)
- Debugger peut être lent

Synthèse:

Idéale pour debugger les structures de données complexes (pointeurs, ...).

Moins adapté pour le debug d'un code externe.

Méthode de debug

Méthode manuelle (printf)
Debugger à points d'arrêts
→ **Détecteur fuites mémoires**

Debug par debugger : automatique

valgrind ./mon_executable


```c
int main()
{
   int *p;
   int u=0;

   *p=5;

   return 0;
}
```

```
==3048== Memcheck, a memory error detector
==3048== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==3048== Using Valgrind-3.8.0 and LibVEX; rerun with -h for copyright info
==3048== Command: ./a.out
==3048==
==3048== Use of uninitialised value of size 8
==3048==    at 0x4004BB: main (debug_04.c:10)
==3048==
==3048==
==3048== HEAP SUMMARY:
==3048==     in use at exit: 0 bytes in 0 blocks
==3048==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==3048==
==3048== All heap blocks were freed -- no leaks are possible
==3048==
==3048== For counts of detected and suppressed errors, rerun with: -v
==3048== Use --track-origins=yes to see where uninitialised values come from
==3048== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

=> Autrement: non détéctée en tant que seg-fault.

Debug par debugger : automatique

Gestion mémoire dynamique:

```c
int main()
{
  int *p=NULL;
  p=malloc(10);



  return 0;
}
```

```
==3666== Memcheck, a memory error detector
==3666== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==3666== Using Valgrind-3.8.0 and LibVEX; rerun with -h for copyright info
==3666== Command: ./a.out
==3666==
==3666==
==3666== HEAP SUMMARY:
==3666==     in use at exit: 10 bytes in 1 blocks
==3666==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==3666==
==3666== LEAK SUMMARY:
==3666==    definitely lost: 10 bytes in 1 blocks
==3666==    indirectly lost: 0 bytes in 0 blocks
==3666==      possibly lost: 0 bytes in 0 blocks
==3666==    still reachable: 0 bytes in 0 blocks
==3666==         suppressed: 0 bytes in 0 blocks
==3666== Rerun with --leak-check=full to see details of leaked memory
==3666==
==3666== For counts of detected and suppressed errors, rerun with: -v
==3666== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

très difficile à détecter autrement!

```c
int main()
{
  int *p=NULL;
  p=malloc(10);

  free(p);

  return 0;
}
```

```
==3555== Memcheck, a memory error detector
==3555== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==3555== Using Valgrind-3.8.0 and LibVEX; rerun with -h for copyright info
==3555== Command: ./a.out
==3555==
==3555==
==3555== HEAP SUMMARY:
==3555==     in use at exit: 0 bytes in 0 blocks
==3555==   total heap usage: 1 allocs, 1 frees, 10 bytes allocated
==3555==
==3555== All heap blocks were freed -- no leaks are possible
==3555==
==3555== For counts of detected and suppressed errors, rerun with: -v
==3555== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

```c
int main()
{
  int *p=NULL;
  p=malloc(10);

  free(p);
  free(p);
  return 0;
}
```

```
==3615== Memcheck, a memory error detector
==3615== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==3615== Using Valgrind-3.8.0 and LibVEX; rerun with -h for copyright info
==3615== Command: ./a.out
==3615==
==3615== Invalid free() / delete / delete[] / realloc()
==3615==    at 0x4C2ACBC: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3615==    by 0x400571: main (debug_05.c:11)
==3615==  Address 0x51da040 is 0 bytes inside a block of size 10 free'd
==3615==    at 0x4C2ACBC: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3615==    by 0x400565: main (debug_05.c:10)
==3615==
==3615==
==3615== HEAP SUMMARY:
==3615==     in use at exit: 0 bytes in 0 blocks
==3615==   total heap usage: 1 allocs, 2 frees, 10 bytes allocated
==3615==
==3615== All heap blocks were freed -- no leaks are possible
==3615==
==3615== For counts of detected and suppressed errors, rerun with: -v
==3615== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

Debug par debugger : automatique

Outils:

Valgrind

émule le processeur+registres
=> vérifie l'utilisation mémoire

Fonctionnement:

valgrind ./mon_executable

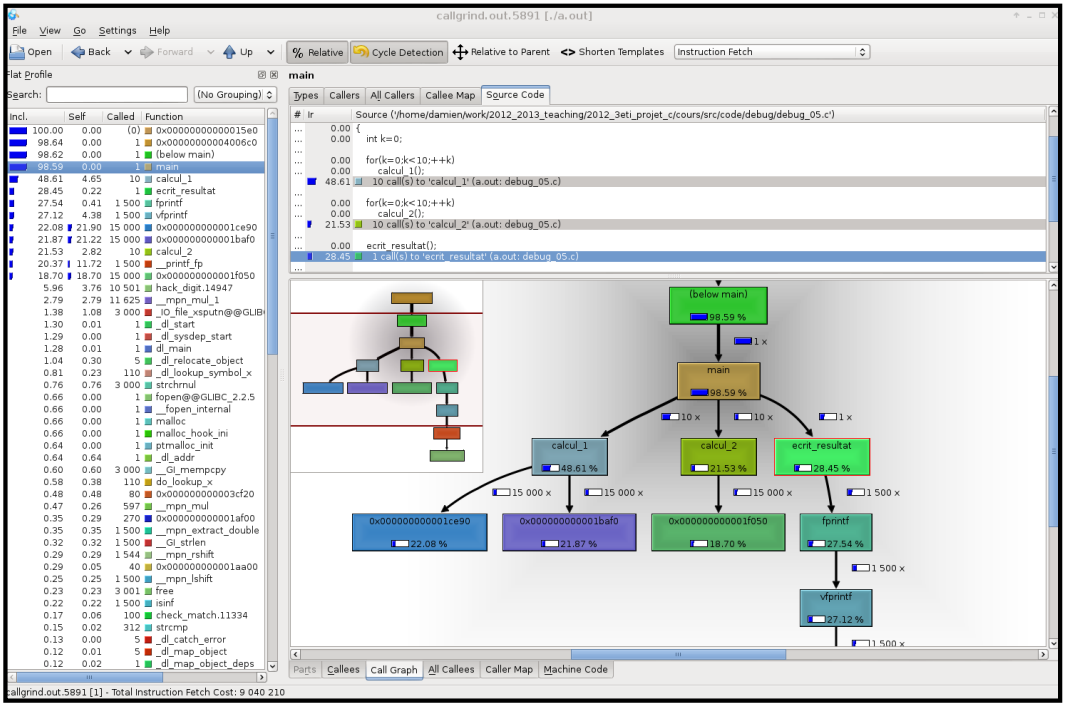
compilé en mode debug

Très utile pour du controle final (*sanity check*)

Debug par debugger : automatique

Possibilité de profiling


```
valgrind --tool=cachegrind ./mon_executable
kcachegrind callgrind.out.xxxx
```



Debug par debugger : automatique

Avantages:

+ détection automatique erreurs
+ permet d'assurer l'absence de fuites mémoire

Inconvénients:

- uniquement debug erreurs mémoires
- code émulé
  (plus lent, comportement différent possible)
- détection d'erreurs librairies externes

Synthèse:

Très adapté pour la détection de fuite mémoire pour un grand code.
Il n'est pas nécessaire d'être familier du code.

Toujours utiliser en tant que "*sanity check*" avant de délivrer le code.

Sortie parfois difficile à interpréter, surtout si d'autres librairies possèdent des fuites mémoires elle-même.

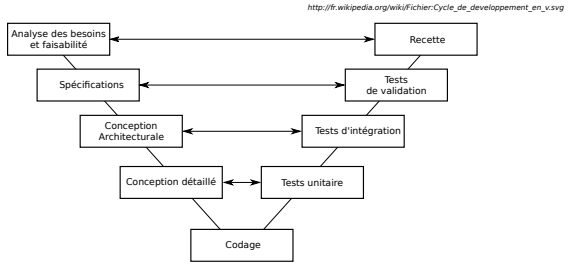
Tests

Principe
Méthodologies de tests unitaires

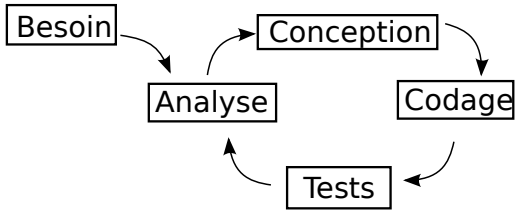
Methodes de génie logiciel

Grandes méthodologies

Approche en V

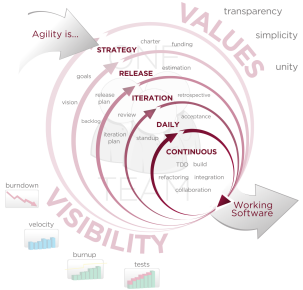


Approche itérative



Approche Agile

Extreme Programming (XP)
Agle Unified Process (AUP)
Dynamic Systems Development Methode (DSDM)
Feature Driven Development (FDD)
Scrum
Kanban
...



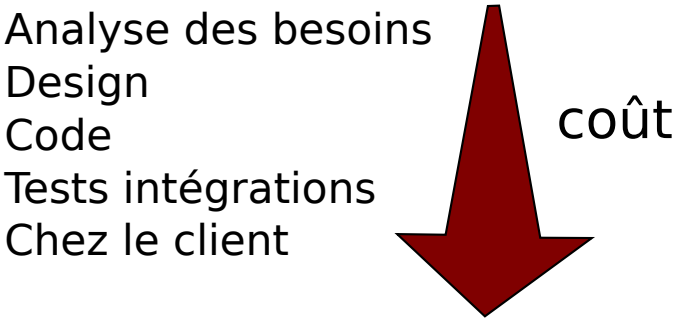
Role des tests

Importance capitale des tests !

Code de qualité =>
Tests permettant de certifier le code

Tester tout au long du developpement

But: détecter les defauts le plus tôt possible



Catégories des tests

Tests unitaires	Test de fonctionalitées de manière unitaire. (plusieurs tests par fonctions: très nombreux)
Tests d'intégrations	Tests la bonne execution d'un ensemble de fonctionnalités (plusieurs tests par grande fonctionalité)
Tests système	Test finale du système au complet (quelques tests par logiciel)

Principe des tests

Dans un logiciel de qualité
=> Toute les fonctions doivent être
testée de manière unitaire

Testez un maximum vos fonctions!

Les cas de bases
Les cas avancés

Les cas qui fonctionnent
Les cas qui ne fonctionnent pas
Les cas critiques/particuliers

Le plus exhaustivement possible

Tests

Principe

→ **Méthodologies de tests unitaires**

Tests unitaires

Tests par échantillonnage

```
int somme(int a,int b);

int test_somme()
{
  if(somme(1,1)!=2)
    ERREUR_TEST;
  if(somme(5,8)!=13)
    ERREUR_TEST;
  if(somme(-5,8)!=3)
    ERREUR_TEST;
  if(somme(-10,-5)!=-15)
    ERREUR_TEST;
}
```

cas base

cas normal

nombre négatif

nombre négatif x2

Tests unitaires

Tests par échantillonnage (avec cas particuliers)

```c
#define GRAND_ENTIER 2147483647
int somme(int a,int b,int *code_erreur);

int test_somme()
{
  int code_erreur=0;
  if(somme(1,1,&code_erreur)!=2 || code_erreur!=0)
    ERREUR_TEST;
  if(somme(5,8,&code_erreur)!=13 || code_erreur!=0)
    ERREUR_TEST;
  if(somme(-5,8,&code_erreur)!=3 || code_erreur!=0)
    ERREUR_TEST;
  if(somme(-10,-5,&code_erreur)!=-15 || code_erreur!=0)
    ERREUR_TEST;
  if(somme(0,1,&code_erreur)!=1 || code_erreur!=0)
    ERREUR_TEST;
  if(somme(0,0,&code_erreur)!=0 || code_erreur!=0)
    ERREUR_TEST;

  int code_erreur=0;
  somme(GRAND_ENTIER,1,&code_erreur);
  if(code_erreur!=ERREUR_DEPASSEMENT)
    ERREUR_TEST;

  int code_erreur=0;
  somme(1,GRAND_ENTIER,&code_erreur);
  if(code_erreur!=ERREUR_DEPASSEMENT)
    ERREUR_TEST;

  int code_erreur=0;
  somme(-GRAND_ENTIER,2,&code_erreur);
  if(code_erreur!=ERREUR_DEPASSEMENT)
    ERREUR_TEST;

  int code_erreur=0;
  somme(2,-GRAND_ENTIER,&code_erreur);
  if(code_erreur!=ERREUR_DEPASSEMENT)
    ERREUR_TEST;

  return OK_TEST;
}
```

test avec le plus grand entier

=> Nécessite de gérer une erreur

=> implémentation + complexe que return a+b;

Tests unitaires

Tests automatiques

```
int somme(int a,int b);

int test_somme()
{
  int nombre_max=9999999;

  int k1=-nombre_max;
  int k2=-nombre_max;
  for(k1=0;k1<nombre_max;++k1)
    for(k2=0;k2<nombre_max;++k2)
      if(somme(k1,k2) != k1+k2)
    ERREUR_TEST;

  return TEST_OK;
}
```

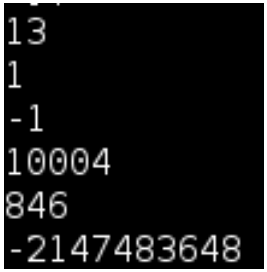
=> Permet d'être plus exhaustif

Tests unitaires

Gestion de l'affichage

ex. de mauvais affichage:

```
printf("%d\n",somme(5,8));
printf("%d\n",somme(0,1));
printf("%d\n",somme(-1,0));
printf("%d\n",somme(9999,5));
printf("%d\n",somme(845,1));
printf("%d\n",somme(2147483647,1));
```



long et difficile à interpréter!

But=> faciliter l'analyse pour pouvoir lancer les tests plusieurs fois.
Préférez solution type: **OK** / **KO**

Tests unitaires

Gestion de l'affichage

Amélioration:

```c
#define TEST_ECHOUE printf("Test unitaire echoue: l.%d fichier %s\n",__LINE__,__FILE__)

int somme(int a,int b);

void test_somme()
{
    if(somme(1,1)!=2)
        TEST_ECHOUE;
    if(somme(1,5)!=6)
        TEST_ECHOUE;
    if(somme(0,-1)!=-1)
        TEST_ECHOUE;
    if(somme(-4,-12)!=-16)
        TEST_ECHOUE;
}

int main()
{
    test_somme();
    return 0;
}

int somme(int a,int b)
{
    if(b<0) a+=1;
    return a+b;
}
```

Affichage uniquement en cas d'erreur.
Indique: ligne+fichier

Test unitaire echoue: l.13 fichier test_unit.c
Test unitaire echoue: l.15 fichier test_unit.c

Tests unitaires

Gestion de l'affichage

Autre possibilité: assert

```c
int somme(int a,int b);

void test_somme()
{
    assert(somme(1,1)==2);
    assert(somme(1,5)==6);
    assert(somme(0,-1)==-1);
    assert(somme(-4,-12)==-16);
}

int main()
{
    test_somme();
    return 0;
}

int somme(int a,int b)
{
    if(b<0) a+=1;
    return a+b;
}
```

Affichage uniquement en cas d'erreur.
S'arrête à la première erreur
Indique: ligne+fichier+condition en erreur

```
a.out: test_03.c:11: test_somme: Assertion `somme(0,-1)==-1' failed.
Aborted
```

Gestion de l'affichage

Affichage complet

```c
#define UNIT_TEST(commande) \
    if( !(commande) ) \
     {printf("Test echoue: [%s], l.%d, fonction %s, fichier %s\n", \
                  #commande,__LINE__,__FUNCTION__,__FILE__);} \
    else \
     {printf("Test OK: [%s]\n",#commande);} \


int somme(int a,int b);

void test_somme()
{
    UNIT_TEST(somme(1,1)==2);
    UNIT_TEST(somme(1,5)==6);
    UNIT_TEST(somme(10,5)==15);
    UNIT_TEST(somme(-1,5)==4);
    UNIT_TEST(somme(-1,-5)==-6);
    UNIT_TEST(somme(-100,5)==-95);
}

int main()
{
    test_somme();
    return 0;
}

int somme(int a,int b)
{
    if(b<0) a+=1;
    return a+b;
}
```

```
Test OK: [somme(1,1)==2]
Test OK: [somme(1,5)==6]
Test OK: [somme(10,5)==15]
Test OK: [somme(-1,5)==4]
Test echoue: [somme(-1,-5)==-6], l.21, fonction test_somme, fichier test_03.c
Test OK: [somme(-100,5)==-95]
```

Tests unitaires

Synthèse:

L'écriture des tests est un travail long

Souvent + long que le code lui même

Ecriture
+
Reflexion sur choix pertinent de tests

Mais: Gain de temps au final

Moins de debug
Pas de retour en arrière

=> Code de qualité

Ecrit 1x
Utilisé un grand nombre de fois

Un bon test = test qui se réutilise
test dont le résultat est rapidement analysé
test qui couvre un grand nombre de cas (corrects et incorrects)
test qui vérifie les cas critiques

Tests unitaires

Outils de tests unitaires:

Macros C (bon cas d'utilisation de macros)

CUnit

http://cunit.sourceforge.net/

```c
/* Simple test of fprintf().
 * Writes test data to the temporary file and checks
 * whether the expected number of bytes were written.
 */
void testFPRINTF(void)
{
   int i1 = 10;

   if (NULL != temp_file) {
      CU_ASSERT(0 == fprintf(temp_file, ""));
      CU_ASSERT(2 == fprintf(temp_file, "Q\n"));
      CU_ASSERT(7 == fprintf(temp_file, "i1 = %d", i1));
   }
}
```

tableau de bord des résultats

Design d'API

Design code de logiciel

Fonction: 2 utilitées

1. Effectuer une tache complexe pour le programmeur
2. Proposer une interface simple pour l'*utilisateur*

algorithmique

ex.

```
float calcul_volume_seve(const struct *tronc,
                         const struct *feuillage,
                         int age);
```

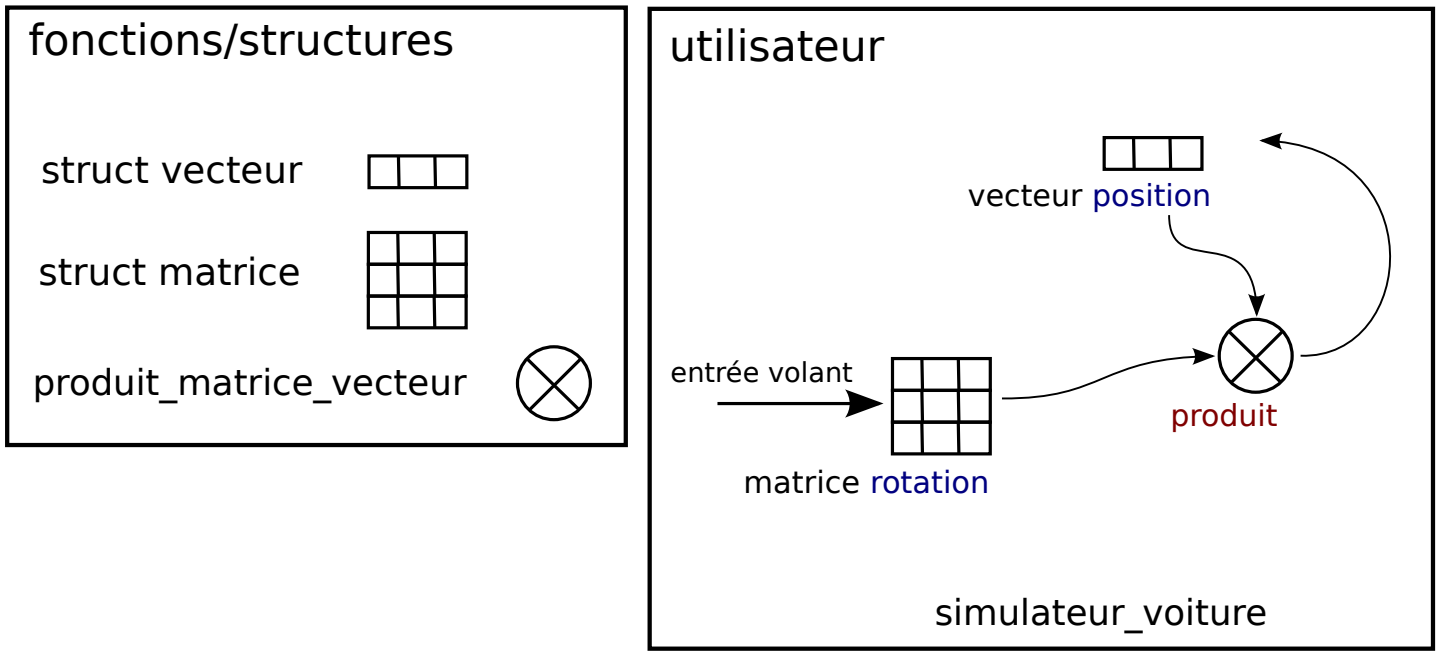
design API

```
struct foret genere_foret();
```

Design code de logiciel

2. Proposer une interface simple pour l'*utilisateur*

Personne (programmeur) qui va utiliser les éléments du code
en tant que boite noire pour réaliser une tache plus complexe.

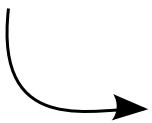


Design code de logiciel

Design d'API =
comment réaliser des fonctions/structures utilisable aisément par l'utilisateur

API = Application Programming Interface
=> L'interface permettant d'utiliser le code de l'application

En C: Design d'API = Design des en-têtes de fonctions + structs !



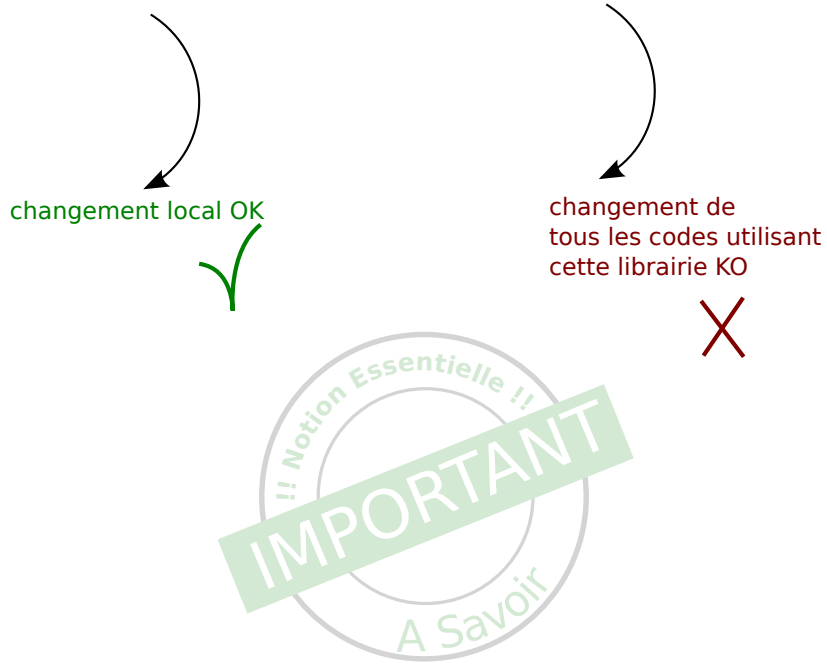
L'API est donnée par les fichiers .h

Note: Toutes les en-têtes (fichiers .h) ne font pas partit de l'API

Design API

Règles standards d'utilisation:

l'API doit être fixe au cours du temps
(l'implémentation peut changer, pas l'API)



Design API

Règles standards d'utilisation: l'API doit être fixe au cours du temps
(l'implémentation peut changer, pas l'API)

exemple:

```
struct vecteur;

void vecteur_init(struct vecteur* v,float x,float y,float z);
float vecteur_norme(const struct vecteur* v);
```

interface

```
int main()
{
  struct vecteur v0;
  struct vecteur v1;
  struct vecteur v2;

  vecteur_init(&v0,0,1,0);
  vecteur_init(&v1,1,1,0);
  vecteur_init(&v2,0,1,2);

  float n0=vecteur_norme(&v0);
  float n1=vecteur_norme(&v1);
  float n2=vecteur_norme(&v2);

  return 0;
}
```

utilisation

Design API

Règles standards d'utilisation: l'API doit être fixe au cours du temps
(l'implémentation peut changer, pas l'API)

exemple:

```
struct vecteur;

void vecteur_init(struct vecteur* v,float x,float y,float z);
float vecteur_norme(const struct vecteur* v);
```

interface

+ L'interface et l'utilisation sont identiques!!!

```
int main()
{
  struct vecteur v0;
  struct vecteur v1;
  struct vecteur v2;

  vecteur_init(&v0,0,1,0);
  vecteur_init(&v1,1,1,0);
  vecteur_init(&v2,0,1,2);

  float n0=vecteur_norme(&v0);
  float n1=vecteur_norme(&v1);
  float n2=vecteur_norme(&v2);

  return 0;
}
```

utilisation

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

void vecteur_init(struct vecteur* v,float x,float y,float z)
{
  v->x=x;
  v->y=y;
  v->z=z;
}
float vecteur_norme(const struct vecteur* v)
{
  return sqrt(v->x*v->x+v->y*v->y+v->z*v->z);
}
```

implémentation 1

```
struct vecteur
{
  float donnees[3];
};
void vecteur_init(struct vecteur* v,float x,float y,float z)
{
  v->donnees[0]=x;
  v->donnees[1]=y;
  v->donnees[2]=z;
}
float vecteur_norme(const struct vecteur* v)
{
  float n2=v->donnees[0]*v->donnees[0]+
    v->donnees[1]*v->donnees[1]+
    v->donnees[2]*v->donnees[2];
  return sqrt(n2);
}
```

implémentation 2

Design API

Règles standards d'utilisation: l'API doit être fixe au cours du temps
(l'implémentation peut changer, pas l'API)

exemple:

```
struct vecteur;

void vecteur_init(struct vecteur* v,float x,float y,float z);
float vecteur_norme(const struct vecteur* v);
```

interface 1

```
int main()
{
  struct vecteur v0;
  struct vecteur v1;
  struct vecteur v2;

  vecteur_init(&v0,0,1,0);
  vecteur_init(&v1,1,1,0);
  vecteur_init(&v2,0,1,2);

  float n0=vecteur_norme(&v0);
  float n1=vecteur_norme(&v1);
  float n2=vecteur_norme(&v2);

  return 0;
}
```

utilisation

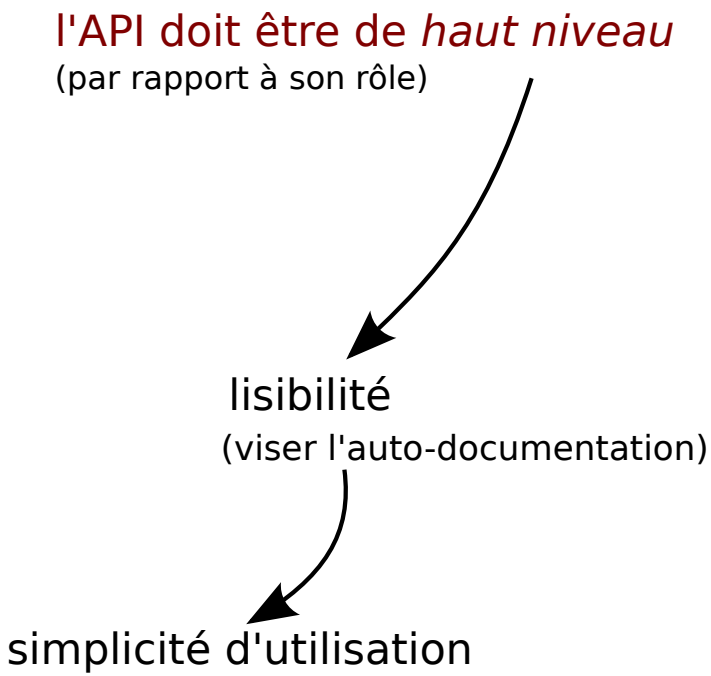
```
struct vecteur;

void vecteur_init(struct vecteur* v,float donnees[3]);
void vecteur_norme(const struct vecteur* v,float *norme);
```

interface 2

- pas compatible!!!
Modification ensemble du programme nécessaire!

Design API

Règles standards d'utilisation:



Design API

Règles standards d'utilisation:

l'API doit être de *haut niveau*
(par rapport à son rôle)

API

```c
#define TAILLE_DATE 12
#define TAILLE_NOM 128
#define TAILLE_DONNEES 4096

struct ip
{
  int donnees[4];
};
struct message
{
  struct ip ip_source;
  struct ip ip_destination;
  char auteur[TAILLE_NOM];
  char donnees[TAILLE_DONNEES];
  char date[TAILLE_DATE];
};
void message_initialise(struct message* mon_message,
                        struct ip ip_source,
                        struct ip ip_destination,
                        char auteur[],
                        char donnees[],
                        char date[]);


struct modem;
void modem_connecte_internet(struct modem* mon_modem);
void modem_fin_connection(struct modem* mon_modem);


struct message recoit_message(const struct modem* mon_modem);
void envoit_message(const struct modem* mon_modem,
                    const struct message* mon_message);
```

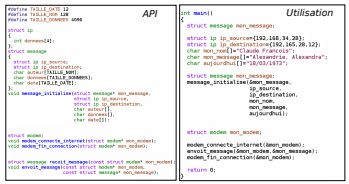
Utilisation

```c
int main()
{
  struct message mon_message;

  struct ip ip_source={192,168,34,28};
  struct ip ip_destination={192,165,28,12};
  char mon_nom[]="Claude Francois";
  char mon_message[]="Alexandrie, Alexandra";
  char aujourdhui[]="18/03/1973";

  struct message mon_message;
  message_initialise(&mon_message,
                     ip_source,
                     ip_destination,
                     mon_nom,
                     mon_message,
                     aujourdhui);


  struct modem mon_modem;

  modem_connecte_internet(&mon_modem);
  envoit_message(&mon_modem,&mon_message);
  modem_fin_connection(&mon_modem);

  return 0;
}
```

Haut niveau = précis,lisible

Design API

Règles standards d'utilisation:



l'API doit être de *haut niveau*
(par rapport à son rôle)

API

```
struct data
{
  int t_int[2];
  char t_char[3][4096];
};

struct FILE* ipfopen(const char* filename,int mode,int delay);
int ipfwrite(const FILE* fid,void* data,int size);
int ipfclose(const FILE* fid,int delay);
```

Utilisation

```
int main()
{
  struct data msg={{1921683428,1921652812},
                   {"Claude Francois",
                    "Alexandrie, Alexandra",
                    "18/03/1973"}};

  struct FILE *fid=NULL;
  fid=ipfopen("/dev/tty30",R_OPEN|W_OPEN|TCPIP_MODE,12);
  if(fid==NULL)
    {printf("Error fopen\n");abort();}

  ipfwrite(fid,&msg,sizeof(msg));
  ipfclose(fid,12);
  assert(fid==NULL);

  return 0;
}
```

X

Trop bas niveau pour l'application
(serait OK pour lecture/ecriture générique, vue OS)

* peu lisible, nécessite doc complète
* pas de bloc unitaire => difficile à debugger/maintenir
* moins précis => mélange message spécifique avec interface générique

Design API

Règles standards d'utilisation: l'API doit être de *haut niveau*
(par rapport à son rôle)

La notion de haut/bas niveau dépend du contexte/utilisation.

Haut-niveau ~= niveau d'abstraction modélisant l'objet/action réelle
sans autres détails d'implémentations techniques.

```
struct ip_source=struct ip={192,168,34,28};
struct ip_dest_ecran_central=struct ip={192,157,05,01};
struct ip_dest_ecran_droit=struct ip={192,157,05,02};
struct ip_dest_ecran_gauche=struct ip={192,157,05,03};

struct message msg_1={ip_source,
                      ip_dest_ecran_gauche,
                      "inutile",
                      "temps restant: [5:00]\n",
                      "21/04/2012"};
struct message msg_2={ip_source,
                      ip_dest_ecran_central,
                      "inutile",
                      "OM VS PSG\n 0 - 2\n",
                      "21/04/2012"};
struct message msg_3={ip_source,
                      ip_dest_ecran_droit,
                      "inutile",
                      "Applaudissez\n",
                      "21/04/2012"};


struct modem mon_modem;
modem_connecte_internet(&mon_modem);
envoit_message(&mon_modem,&msg_1);
envoit_message(&mon_modem,&msg_2);
envoit_message(&mon_modem,&msg_3);
modem_fin_connection(&mon_modem);
```

trop bas niveau:
inutile d'exposer l'utilisation d'internet

```
struct ecran;
void ecran_initialise(struct ecran* mon_ecran,char identificateur[]);
void ecrit(const struct* ecran,char message[]);
```

nouvelle API au dessus de la gestion de paquets et d'IP

```
struct ecran_central;
struct ecran_lateral_droit;
struct ecran_lateral_gauche;

ecran_initialise(&ecran_central,"centre");
ecran_initialise(&ecran_lateral_gauche,"lateral gauche");
ecran_initialise(&ecran_lateral_droit,"lateral droit");

ecrit(&ecran_lateral_gauche,"temps restant: [5:00]\n");
ecrit(&ecran_central,"OM VS PSG\n 0 - 2\n");
ecrit(&ecrant_lateral_droit,"Applaudissez\n");
```

haut niveau:
on ne gère que l'identifiant des écrans et le texte

Exemple: affichage d'un message sur 3 écrans lors d'un match de foot

Design API

Règles standards d'utilisation:

l'API doit être de *haut niveau*
(par rapport à son rôle)

Rappel: Un bon code = code **lisible**
= code qui **cache sa complexité**
= code qui propose de manipuler des **abstractions** aisément

Design API

Règles standards d'utilisation:

Chaque bloc de l'API doit **minimiser**:

* l'exposition de son implémentation

* le nombre de fonctions les manipulants

* les interactions avec les autres blocs

Truc: Pour vérifier l'organisation de votre interface, représenter la (avec les interactions) sous forme de schéma bloc.
Le schéma est il simple, logique, lisible?

Design API

Règles standards d'utilisation:

minimiser l'exposition de son implémentation

Exemple: accès coordonnées d'un vecteur:

vecteur	vecteur	vecteur
float x float y float z	float d[3]	char d[12]

x=vecteur.x
y=vecteur.y
z=vecteur.z
	x=vecteur.d[0]
	y=vecteur.d[1]
	z=vecteur.d[2]
		x=*(float*)(vecteur.d);
		y=*(float*)(vecteur.d+4);
		z=*(float*)(vecteur.d+8);

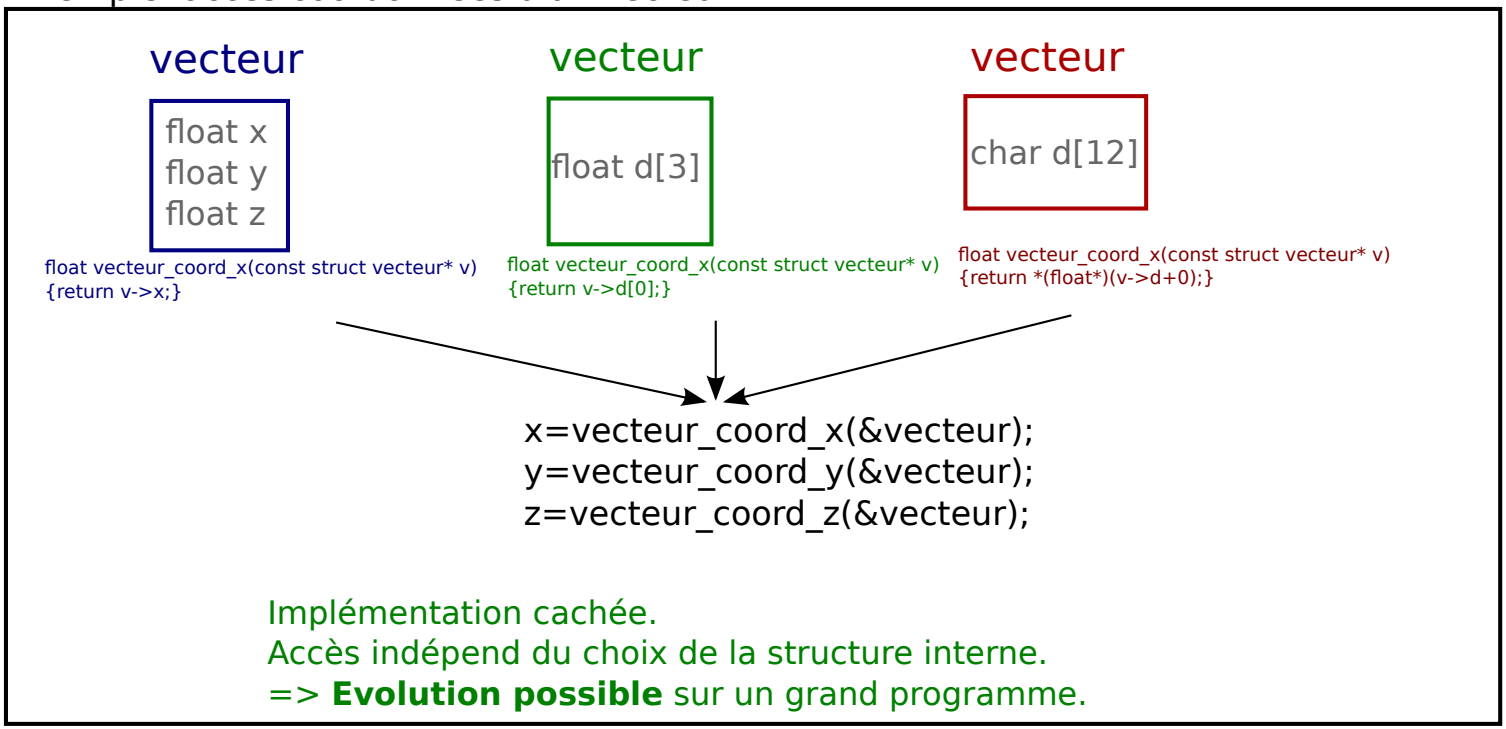
Implémentation non cachée.
Accès dépend du choix de la structure interne.
=> Evolution impossible sur un grand programme.

Design API

Règles standards d'utilisation:

minimiser l'exposition de son implémentation

Exemple: accès coordonnées d'un vecteur:



Design API

Règles standards d'utilisation:

minimiser l'exposition de son implémentation

Note: Pour les struct *conteneurs*

but est de contenir des données
(vecteurs, tableaux, ...)

Opérations de *get/set* très classiques

Dénomination=**Accessors**

ex.

```
vecteur v;
v.get_x();
v.get_y(4.5);
```

```
tree b;
b.get_trunc();
```

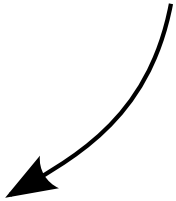
```
airplane a;
a.get_reactor();
a.set_weapon(weapon_1);
```

=> On ne sait pas comment sont stockées les données
=> On ne veut pas le savoir vue utilisateur (encapsulation)
=> Stockage interne peut être modifié sans changer l'API

Design API

Règles standards d'utilisation:

minimiser le nombre de fonctions les manipulants



fonctionnalitées précises
niveau d'abstraction bien défini

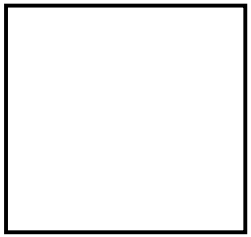
+ simple à gérer
+ simple à faire évoluer
+ augmente la cohérence

Design API

Règles standards d'utilisation:

minimiser le nombre de fonctions les manipulants

struct voiture



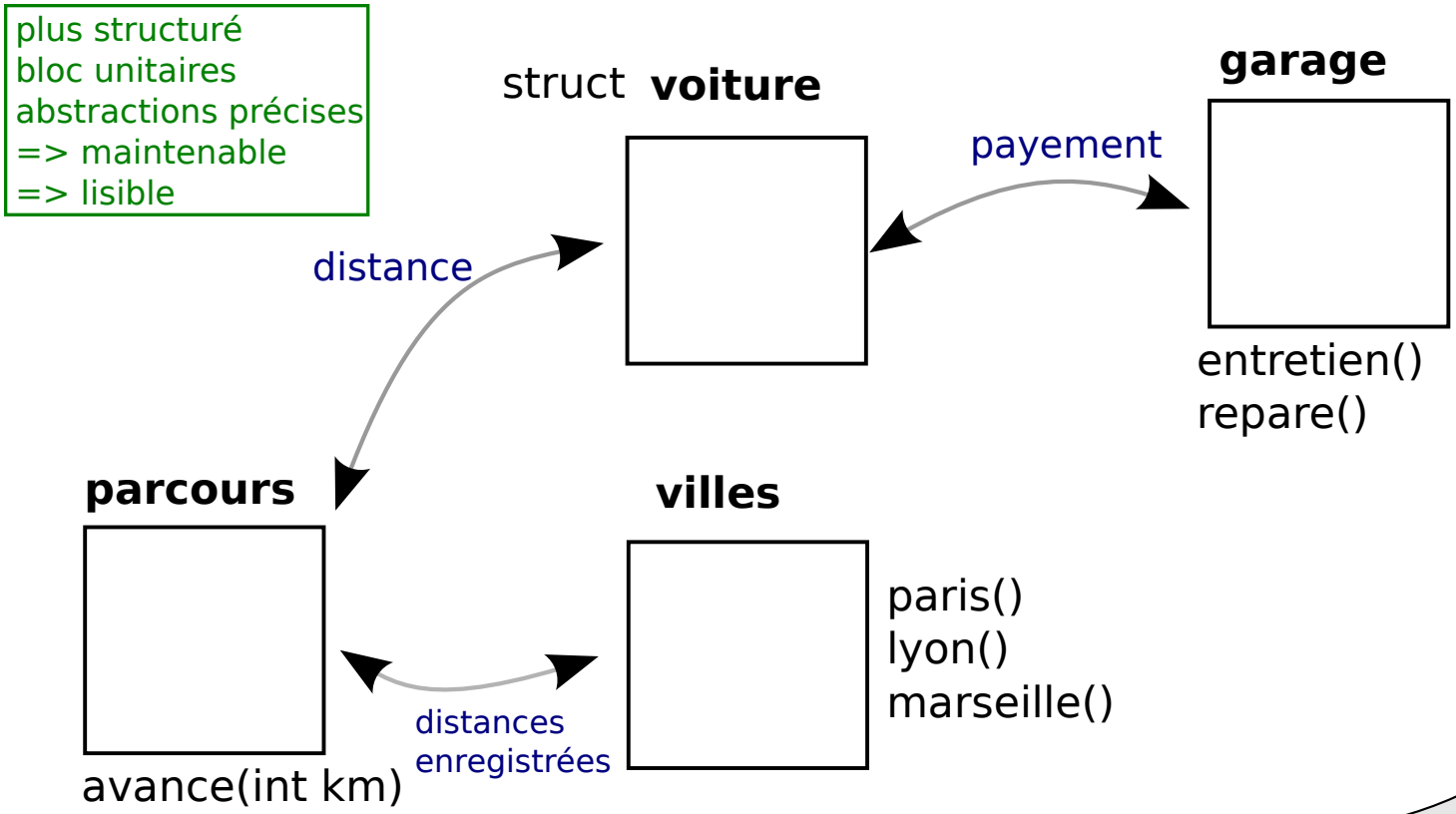
voiture_set/get_carburant();
voiture_set/get_puissance();
voiture_distance_parcourus();
voiture_avance(int km);
voiture_avance_paris();
voiture_avance_marseille();
voiture_avance_lyon();
voiture_entretien();
voiture_repare();

- trop de fonctions
- difficile à manipuler/faire évoluer

Design API

Règles standards d'utilisation:

minimiser le nombre de fonctions les manipulants

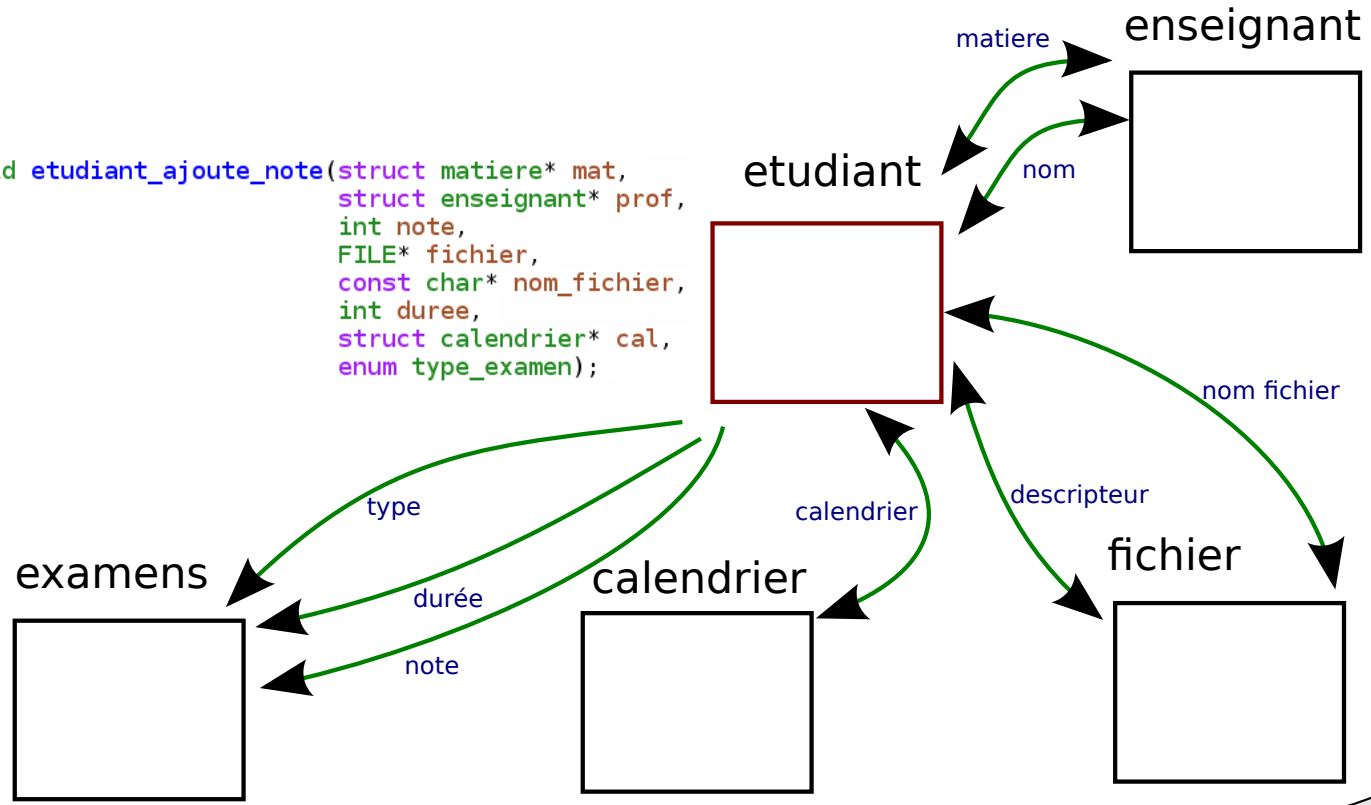


Design API

Règles standards d'utilisation:

minimiser les interactions avec les autres blocs

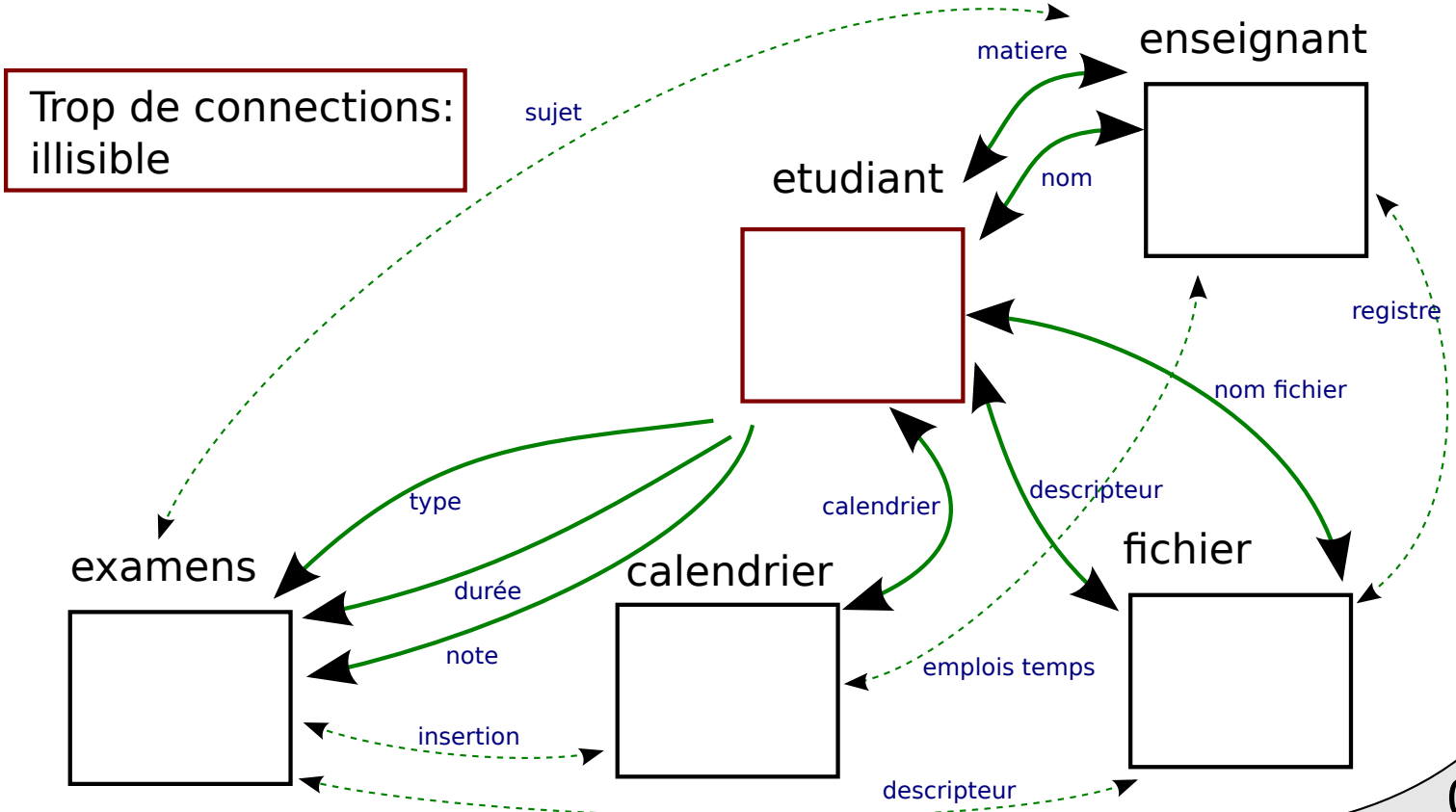
```
void etudiant_ajoute_note(struct matiere* mat,
                          struct enseignant* prof,
                          int note,
                          FILE* fichier,
                          const char* nom_fichier,
                          int duree,
                          struct calendrier* cal,
                          enum type_examen);
```



Design API

Règles standards d'utilisation:

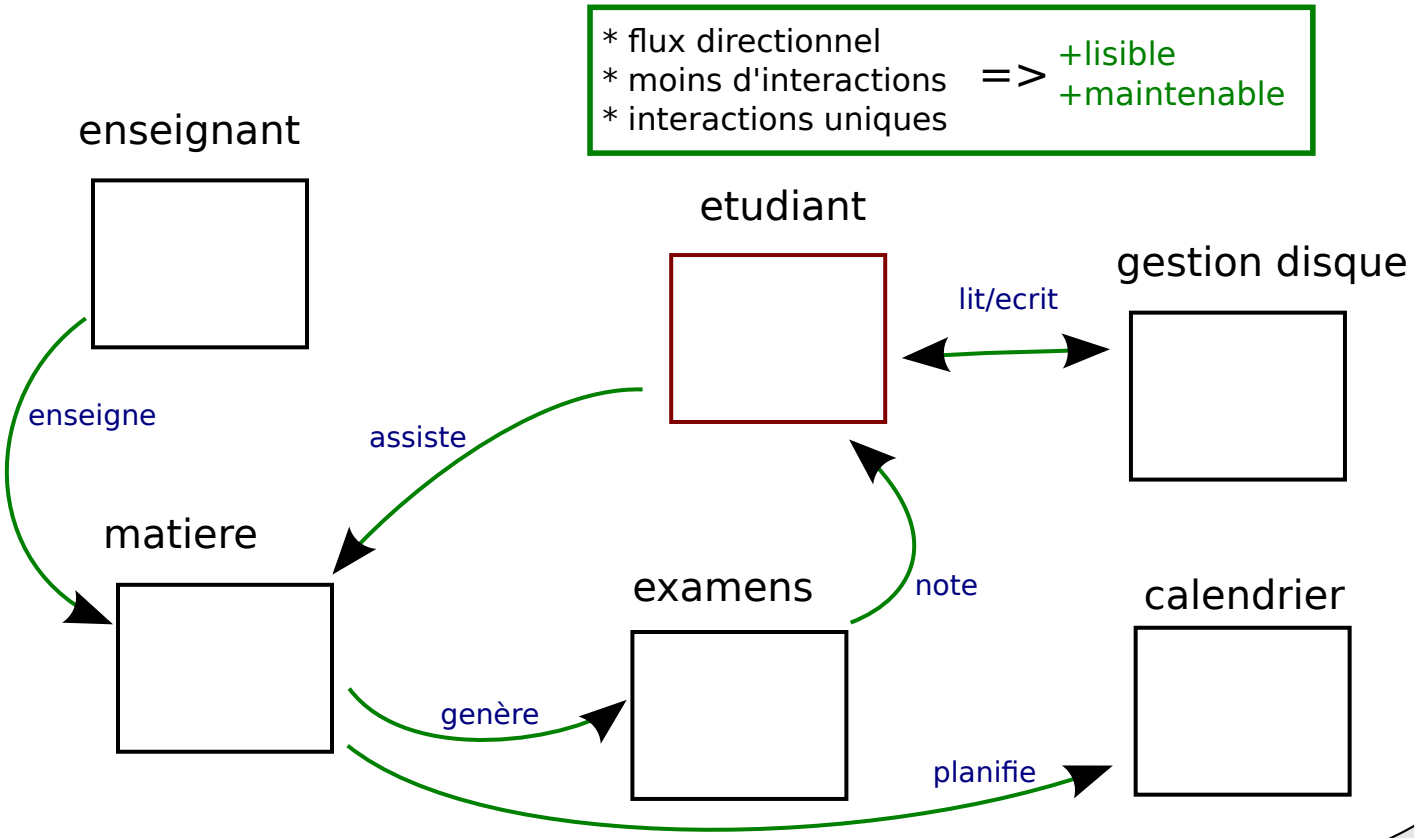
minimiser les interactions avec les autres blocs



Design API

Règles standards d'utilisation:

minimiser les interactions avec les autres blocs



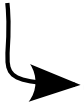
Design API

Règles standards d'utilisation:

minimiser les interactions avec les autres blocs

Règles suggérées:

1 bloc communique avec au plus 3 autres blocs
Evitez les communication bi-directionnelles
Evitez les abstractions *omniscientes*



qui a accès à tout