Macro-instructions de contrôle de compilation

Les macro-instructions de contrôle de compilation permettent d'ignorer, lors de la compilation, une partie du programme.

Ceci est utile pour **sélectionner** les parties à prendre en compte dans un programme, sans avoir à les (dé)commenter.

Macro-instructions de contrôle de compilation

Les macro-instructions de contrôle de compilation permettent d'ignorer, lors de la compilation, une partie du programme.

Ceci est utile pour **sélectionner** les parties à prendre en compte dans un programme, sans avoir à les (dé)commenter.

On dispose ainsi des constructions

```
#ifdef SYMB
...
#endif
```

```
#ifndef SYMB
...
#endif
```

```
#ifdef SYMB
...
#else
...
#endif
```

À gauche, le code `...` n'est considéré que si l'alias `SYMB` est défini.

Au centre, le code `...` n'est considéré que si l'alias `SYMB` n'est pas défini.

Macro-instructions de contrôle de compilation

```
#include <stdio.h>


#define GAUCHE_DROITE

#ifdef GAUCHE_DROITE

int rechercher(char *tab,
        int n, char x) {
    int i;
    for (i = 0 ; i < n ; ++i)
        if (tab[i] == x)
            return i;
    return -1;
}

#else

int rechercher(char *tab,
        int n, char x) {
    int i;
    for (i = n - 1 ; i >= 0 ; -i)
        if (tab[i] == x)
            return i;
    return -1;
}

#endif

int main() {
    int res;
    char tab[] = "chaine de test";

    res = rechercher(tab, 14, 't');
    printf("%d\n", res);

}
```

Ici, on donne deux algorithmes pour localiser la première occurrence d'une lettre dans un tableau : de la gauche vers la droite, ou bien de la droite vers la gauche.

Ce programme affiche `10` ; si on renomme `GAUCHE_DROITE` (en l. 4), il affiche `13`.

Plan

Plan

Mise en page d'un programme

Pour écrire un programme de valeur, il faut soigner les points suivants :

1. l'indentation ;
2. l'organisation des espaces autour des caractères ;
3. le choix des identificateurs ;
4. la documentation.

Indentation

L'indentation consiste à disposer des caractères blancs au début de certaines lignes d'un programme.

Contrairement au Python, celle-ci ne modifie pas le comportement d'un programme.

L'objectif est d'augmenter sa lisibilité.

Indentation

L'indentation consiste à disposer des caractères blancs au début de certaines lignes d'un programme.

Contrairement au Python, celle-ci ne modifie pas le comportement d'un programme.

L'objectif est d'augmenter sa lisibilité.

Règle : on place **quatre espaces** (pas de tabulation) avant chaque instruction d'un bloc et on incrémente cet espacement de quatre en quatre en fonction de la profondeur des blocs.

Indentation

L'indentation consiste à disposer des caractères blancs au début de certaines lignes d'un programme.

Contrairement au Python, celle-ci ne modifie pas le comportement d'un programme.

L'objectif est d'augmenter sa lisibilité.

Règle : on place **quatre espaces** (pas de tabulation) avant chaque instruction d'un bloc et on incrémente cet espacement de quatre en quatre en fonction de la profondeur des blocs.

```
/* Correct. */
a␣=␣8;
if␣(b␣>=␣0)␣{
␣␣␣␣printf("%d\n",␣a);
␣␣␣␣a␣=␣0;
␣␣␣␣for␣(i␣=␣0␣;␣i␣<=␣b␣;␣++i)
␣␣␣␣␣␣␣␣a␣/=␣2;
}
```

Indentation

L'indentation consiste à disposer des caractères blancs au début de certaines lignes d'un programme.

Contrairement au Python, celle-ci ne modifie pas le comportement d'un programme.

L'objectif est d'augmenter sa lisibilité.

Règle : on place **quatre espaces** (pas de tabulation) avant chaque instruction d'un bloc et on incrémente cet espacement de quatre en quatre en fonction de la profondeur des blocs.

```
/* Correct. */
a␣=␣8;
if␣(b␣>=␣0)␣{
␣␣␣␣printf("%d\n",␣a);
␣␣␣␣a␣=␣0;
␣␣␣␣for␣(i␣=␣0␣;␣i␣<=␣b␣;␣++i)
␣␣␣␣␣␣␣␣a␣/=␣2;
}
```

```
/* Incorrect. */
a␣=␣8;
if␣(b␣>=␣0)␣{
␣␣printf("%d\n",␣a);
␣␣a␣=␣0;
␣␣for␣(i␣=␣0␣;␣i␣<=␣b␣;␣++i)
␣␣␣␣␣␣a␣/=␣2;
}
```

Organisation des blocs

Nous avons vu qu'un bloc des une suite d'instructions délimitée par des accolades. Il se trouve attaché à une instruction de branchement ou de boucle. Il peut aussi être indépendant.

On **revient à la ligne** après une **accolade ouvrante**.

L'**accolade fermante** se trouve horizontalement au **niveau du début** de la ligne qui contient l'accolade ouvrante.

Organisation des blocs

Nous avons vu qu'un bloc des une suite d'instructions délimitée par des accolades. Il se trouve attaché à une instruction de branchement ou de boucle. Il peut aussi être indépendant.

On **revient à la ligne** après une **accolade ouvrante**.

L'**accolade fermante** se trouve horizontalement au **niveau du début** de la ligne qui contient l'accolade ouvrante.

```
/* Correct. */
if (valeur >= 1) {
    valeur -= 1;
}
```

Organisation des blocs

Nous avons vu qu'un bloc des une suite d'instructions délimitée par des accolades. Il se trouve attaché à une instruction de branchement ou de boucle. Il peut aussi être indépendant.

On **revient à la ligne** après une **accolade ouvrante**.

L'**accolade fermante** se trouve horizontalement au **niveau du début** de la ligne qui contient l'accolade ouvrante.

```
/* Correct. */
if (valeur >= 1) {
    valeur -= 1;
}
```

```
/* Incorrect. */
if (valeur >= 1)
{
    valeur -= 1;
}
```

Organisation des blocs

Nous avons vu qu'un bloc des une suite d'instructions délimitée par des accolades. Il se trouve attaché à une instruction de branchement ou de boucle. Il peut aussi être indépendant.

On **revient à la ligne** après une **accolade ouvrante**.

L'**accolade fermante** se trouve horizontalement au **niveau du début** de la ligne qui contient l'accolade ouvrante.

```
/* Correct. */
if (valeur >= 1) {
    valeur -= 1;
}
```

```
/* Incorrect. */
if (valeur >= 1)
{
    valeur -= 1;
}
```

```
/* Correct. */
valeur = 1;
{
    int a;
    valeur = 10;
}
```

Organisation des blocs

Nous avons vu qu'un bloc des une suite d'instructions délimitée par des accolades. Il se trouve attaché à une instruction de branchement ou de boucle. Il peut aussi être indépendant.

On **revient à la ligne** après une **accolade ouvrante**.

L'**accolade fermante** se trouve horizontalement au **niveau du début** de la ligne qui contient l'accolade ouvrante.

```
/* Correct. */
if (valeur >= 1) {
    valeur -= 1;
}
```

```
/* Incorrect. */
if (valeur >= 1)
{
    valeur -= 1;
}
```

```
/* Correct. */
valeur = 1;
{
    int a;
    valeur = 10;
}
```

```
/* Incorrect. */
valeur = 1; {
    int a;
    valeur = 10;
}
```

Organisation des espaces

On place une **espace avant et après** chaque opérateur.

```
/* Correct. */
a␣=␣b␣*␣2␣+␣5;
```

Organisation des espaces

On place une **espace avant et après** chaque opérateur.

```
/* Correct. */
a␣=␣b␣*␣2␣+␣5;
```

```
/* Incorrect. */
a␣=␣b*2␣+␣5;
```

Organisation des espaces

On place une **espace avant et après** chaque opérateur.

```
/* Correct. */
a␣=␣b␣*␣2␣+␣5;
```

```
/* Incorrect. */
a␣=␣b*2␣+␣5;
```

On utilise les règles habituelles de **typographie** pour l'usage des **virgules**.

```
/* Correct. */
f(a,␣b,␣c,␣16);
```

Organisation des espaces

On place une **espace avant et après** chaque opérateur.

```
/* Correct. */
a␣=␣b␣*␣2␣+␣5;
```

```
/* Incorrect. */
a␣=␣b*2␣+␣5;
```

On utilise les règles habituelles de **typographie** pour l'usage des **virgules**.

```
/* Correct. */
f(a,␣b,␣c,␣16);
```

```
/* Incorrect. */
f(a,b,c,16);
```

Organisation des espaces

On place une **espace avant et après** chaque opérateur.

```
/* Correct. */
a␣=␣b␣*␣2␣+␣5;
```

```
/* Incorrect. */
a␣=␣b*2␣+␣5;
```

On utilise les règles habituelles de **typographie** pour l'usage des **virgules**.

```
/* Correct. */
f(a,␣b,␣c,␣16);
```

```
/* Incorrect. */
f(a,b,c,16);
```

On ne place **pas d'espace** après une **parenthèse ouvrante** ou avant une **parenthèse fermante**.

```
/* Correct. */
a␣=␣(f(a,␣3)␣+␣a)␣*␣2;
```

Organisation des espaces

On place une **espace avant et après** chaque opérateur.

```
/* Correct. */
a␣=␣b␣*␣2␣+␣5;
```

```
/* Incorrect. */
a␣=␣b*2␣+␣5;
```

On utilise les règles habituelles de **typographie** pour l'usage des **virgules**.

```
/* Correct. */
f(a,␣b,␣c,␣16);
```

```
/* Incorrect. */
f(a,b,c,16);
```

On ne place **pas d'espace** après une **parenthèse ouvrante** ou avant une **parenthèse fermante**.

```
/* Correct. */
a␣=␣(f(a,␣3)␣+␣a)␣*␣2;
```

```
/* Incorrect. */
a␣=␣(f(␣a,␣3)␣+␣a␣)␣*␣2;
```

Choix des identificateurs

Les identificateurs doivent à la fois **renseigner sur le rôle** des entités auxquelles ils appartiennent (variables, paramètres, fonctions, modules, *etc.*) et être **concis**.

Choix des identificateurs

Les identificateurs doivent à la fois **renseigner sur le rôle** des entités auxquelles ils appartiennent (variables, paramètres, fonctions, modules, *etc.*) et être **concis**.

```
/* Identificateur non explicite. */
v
```

Choix des identificateurs

Les identificateurs doivent à la fois **renseigner sur le rôle** des entités auxquelles ils appartiennent (variables, paramètres, fonctions, modules, *etc.*) et être **concis**.

```
/* Identificateur non explicite. */
v

/* Identificateur trop long. */
valeur_choisie_pour_le_nombre_parties
```

Choix des identificateurs

Les identificateurs doivent à la fois **renseigner sur le rôle** des entités auxquelles ils appartiennent (variables, paramètres, fonctions, modules, *etc.*) et être **concis**.

```
/* Identificateur non explicite. */
v

/* Identificateur trop long. */
valeur_choisie_pour_le_nombre_parties

/* Identificateur acceptable. */
nb_parties
```

Choix des identificateurs

Les identificateurs doivent à la fois **renseigner sur le rôle** des entités auxquelles ils appartiennent (variables, paramètres, fonctions, modules, *etc.*) et être **concis**.

```
/* Identificateur non explicite. */
v

/* Identificateur trop long. */
valeur_choisie_pour_le_nombre_parties

/* Identificateur acceptable. */
nb_parties
```

On fixe la langue au **français** pour leur construction.

Choix des identificateurs

Les seuls identificateurs d'une lettre autorisés sont `i`, `j`, `k`, *etc.*, pour les indices de boucles.

Choix des identificateurs

Les seuls identificateurs d'une lettre autorisés sont `i`, `j`, `k`, *etc.*, pour les indices de boucles.

Les majuscules sont interdites dans les identificateurs. Dans un identificateur composé de plusieurs mots, ces derniers sont séparés par des sous-tirets.

```
/* Correct. */
nb_parties
```

```
/* Incorrect. */
nbParties
```

Choix des identificateurs

Les seuls identificateurs d'une lettre autorisés sont `i`, `j`, `k`, *etc.*, pour les indices de boucles.

Les majuscules sont interdites dans les identificateurs. Dans un identificateur composé de plusieurs mots, ces derniers sont séparés par des sous-tirets.

```
/* Correct. */
nb_parties
```

```
/* Incorrect. */
nbParties
```

Exception : les identificateurs de constantes (définies par une instruction pré-processeur) sont écrits exclusivement en majuscules.

```
/* Correct. */
#define TAILLE_MAX 1024
#define DEBUG
```

Documentation

Un programme est documenté par des commentaires. Ce sont des **phrases**, placées entre /* et */.

Documentation

Un programme est documenté par des commentaires. Ce sont des **phrases**, placées entre `/*` et `*/`.

On documente chaque fichier de programme, au tout début, par

- les prénoms et noms des auteurs ;
- la date de création (au format jour-mois-année) ;
- la date de modification (au format précédent).

```
/* Auteur : A. M. Turing
 * Creation : 23-06-1912
 * Modification : 07-06-1954 */
```

Documentation

Un programme est documenté par des commentaires. Ce sont des **phrases**, placées entre `/*` et `*/`.

On documente chaque fichier de programme, au tout début, par

- les prénoms et noms des auteurs ;
- la date de création (au format jour-mois-année) ;
- la date de modification (au format précédent).

```
/* Auteur : A. M. Turing
 * Creation : 23-06-1912
 * Modification : 07-06-1954 */
```

On commentera le moins possible les instructions.

Documentation

Un programme est documenté par des commentaires. Ce sont des **phrases**, placées entre `/*` et `*/`.

On documente chaque fichier de programme, au tout début, par

- les prénoms et noms des auteurs ;
- la date de création (au format jour-mois-année) ;
- la date de modification (au format précédent).

```
/* Auteur : A. M. Turing
 * Creation : 23-06-1912
 * Modification : 07-06-1954 */
```

On commentera le moins possible les instructions.

Il faut éviter les commentaires inutiles.

```
/* Incorrect. */
/* Affiche la valeur de ‘a‘. */
printf("%d\n", a);
```

Documentation des fonctions

On documente la plupart des fonctions par des commentaires situés avant leur déclaration (ou leur définition).

Documentation des fonctions

On documente la plupart des fonctions par des commentaires situés avant leur déclaration (ou leur définition).

Un commentaire de fonction explique

- le **rôle** de chaque **paramètre** ;
- ce que **renvoie** la fonction ;
- l'**effet** produit par la fonction.

Documentation des fonctions

On documente la plupart des fonctions par des commentaires situés avant leur déclaration (ou leur définition).

Un commentaire de fonction explique

- le **rôle** de chaque **paramètre** ;
- ce que **renvoie** la fonction ;
- l'**effet** produit par la fonction.

```
/* Correct. */
/* Renvoie le plus grand entier
 * parmi ‘a‘ et ‘b‘. */
int max(int a, int b) {
    if (a >= b)
        return a;
    return b;
}
```

Documentation des fonctions

On documente la plupart des fonctions par des commentaires situés avant leur déclaration (ou leur définition).

Un commentaire de fonction explique

- le **rôle** de chaque **paramètre** ;
- ce que **renvoie** la fonction ;
- l'**effet** produit par la fonction.

```
/* Correct. */
/* Renvoie le plus grand entier
 * parmi ‘a‘ et ‘b‘. */
int max(int a, int b) {
    if (a >= b)
        return a;
    return b;
}
```

```
/* Incorrect. */
/* Calcule le maximum de deux
 * entiers. */
int max(int a, int b) {
    if (a >= b)
        return a;
    return b;
}
```

Plan

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Ceci peut provoquer une erreur provenant principalement

- du **système** (p.ex. lorsqu'il ne parvient pas à satisfaire un `malloc` pour cause de la configuration de la mémoire);

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Ceci peut provoquer une erreur provenant principalement

- du **système** (p.ex. lorsqu'il ne parvient pas à satisfaire un `malloc` pour cause de la configuration de la mémoire);
- de l'**utilisateur** (p.ex. lorsqu'il communique au programme des données inattendues);

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Ceci peut provoquer une erreur provenant principalement

- du **système** (p.ex. lorsqu'il ne parvient pas à satisfaire un `malloc` pour cause de la configuration de la mémoire) ;
- de l'**utilisateur** (p.ex. lorsqu'il communique au programme des données inattendues) ;
- du **programmeur** (p.ex. lorsqu'il appelle une fonction avec des arguments inadéquats qui rendent le calcul impraticable).

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Ceci peut provoquer une erreur provenant principalement

- du **système** (p.ex. lorsqu'il ne parvient pas à satisfaire un `malloc` pour cause de la configuration de la mémoire) ;
- de l'**utilisateur** (p.ex. lorsqu'il communique au programme des données inattendues) ;
- du **programmeur** (p.ex. lorsqu'il appelle une fonction avec des arguments inadéquats qui rendent le calcul impraticable).

Il est nécessaire de

1. pouvoir détecter les erreurs ;

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Ceci peut provoquer une erreur provenant principalement

- du **système** (p.ex. lorsqu'il ne parvient pas à satisfaire un `malloc` pour cause de la configuration de la mémoire) ;
- de l'**utilisateur** (p.ex. lorsqu'il communique au programme des données inattendues) ;
- du **programmeur** (p.ex. lorsqu'il appelle une fonction avec des arguments inadéquats qui rendent le calcul impraticable).

Il est nécessaire de

1. pouvoir détecter les erreurs ;
2. pouvoir remédier aux erreurs quand elles surviennent.

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Ceci peut provoquer une erreur provenant principalement

- du **système** (p.ex. lorsqu'il ne parvient pas à satisfaire un `malloc` pour cause de la configuration de la mémoire) ;
- de l'**utilisateur** (p.ex. lorsqu'il communique au programme des données inattendues) ;
- du **programmeur** (p.ex. lorsqu'il appelle une fonction avec des arguments inadéquats qui rendent le calcul impraticable).

Il est nécessaire de

1. pouvoir détecter les erreurs ;
2. pouvoir remédier aux erreurs quand elles surviennent.

Pour cela, nous allons augmenter l'écriture de fonctions de mécanismes de gestion d'erreur.

Mécanisme de gestion d'erreurs

On écrira la plupart des fonctions selon le schéma suivant :

- le type de retour est `int` et la valeur de retour, le code d'erreur, renseigne si l'exécution de la fonction s'est bien déroulée ;

Mécanisme de gestion d'erreurs

On écrira la plupart des fonctions selon le schéma suivant :

- le type de retour est `int` et la valeur de retour, le code d'erreur, renseigne si l'exécution de la fonction s'est bien déroulée ;
- la (ou les) valeur(s) « renvoyée(s) » par la fonction se fait par un (des) passage(s) par adresse.

Mécanisme de gestion d'erreurs

On écrira la plupart des fonctions selon le schéma suivant :

- le type de retour est `int` et la valeur de retour, le code d'erreur, renseigne si l'exécution de la fonction s'est bien déroulée ;
- la (ou les) valeur(s) « renvoyée(s) » par la fonction se fait par un (des) passage(s) par adresse.

Schématiquement, une telle fonction admet ainsi le prototype

```
int FCT(T1 E1, ..., TN EN,   U1 S1, ..., UK SK);
```

dit prototype standard

Mécanisme de gestion d'erreurs

On écrira la plupart des fonctions selon le schéma suivant :

- le type de retour est `int` et la valeur de retour, le code d'erreur, renseigne si l'exécution de la fonction s'est bien déroulée ;
- la (ou les) valeur(s) « renvoyée(s) » par la fonction se fait par un (des) passage(s) par adresse.

Schématiquement, une telle fonction admet ainsi le prototype

```
int FCT(T1 E1, ..., TN EN,   U1 S1, ..., UK SK);
```

dit prototype standard où

- les `Ei` sont les paramètres d'entrée ;

Mécanisme de gestion d'erreurs

On écrira la plupart des fonctions selon le schéma suivant :

- le type de retour est `int` et la valeur de retour, le code d'erreur, renseigne si l'exécution de la fonction s'est bien déroulée ;
- la (ou les) valeur(s) « renvoyée(s) » par la fonction se fait par un (des) passage(s) par adresse.

Schématiquement, une telle fonction admet ainsi le prototype

```
int FCT(T1 E1, ..., TN EN,   U1 S1, ..., UK SK);
```

dit prototype standard où

- les `Ei` sont les paramètres d'entrée ;
- les `Ti` sont des types (potentiellement des adresses) ;

Mécanisme de gestion d'erreurs

On écrira la plupart des fonctions selon le schéma suivant :

- le type de retour est `int` et la valeur de retour, le code d'erreur, renseigne si l'exécution de la fonction s'est bien déroulée ;
- la (ou les) valeur(s) « renvoyée(s) » par la fonction se fait par un (des) passage(s) par adresse.

Schématiquement, une telle fonction admet ainsi le prototype

```
int FCT(T1 E1, ..., TN EN,   U1 S1, ..., UK SK);
```

dit prototype standard où

- les `Ei` sont les paramètres d'entrée ;
- les `Ti` sont des types (potentiellement des adresses) ;
- les `Sj` sont les paramètres de sortie ;

Mécanisme de gestion d'erreurs

On écrira la plupart des fonctions selon le schéma suivant :

- le type de retour est `int` et la valeur de retour, le code d'erreur, renseigne si l'exécution de la fonction s'est bien déroulée ;
- la (ou les) valeur(s) « renvoyée(s) » par la fonction se fait par un (des) passage(s) par adresse.

Schématiquement, une telle fonction admet ainsi le prototype

```
int FCT(T1 E1, ..., TN EN,   U1 S1, ..., UK SK);
```

dit prototype standard où

- les `Ei` sont les paramètres d'entrée ;
- les `Ti` sont des types (potentiellement des adresses) ;
- les `Sj` sont les paramètres de sortie ;
- les `Uj` sont des types (potentiellement des adresses).

Mécanisme de gestion d'erreurs

Le code d'erreur d'une fonction ayant un prototype standard suit la spécification suivante :

Mécanisme de gestion d'erreurs

Le code d'erreur d'une fonction ayant un prototype standard suit la spécification suivante :

- une **valeur négative ou nulle** renseigne qu'une erreur s'est produite. En général, la fonction renvoie 0 dans ce cas. On peut être plus précis et exprimer une erreur particulière en renvoyant des autres valeurs négatives.

Mécanisme de gestion d'erreurs

Le code d'erreur d'une fonction ayant un prototype standard suit la spécification suivante :

- une **valeur négative ou nulle** renseigne qu'une erreur s'est produite. En général, la fonction renvoie `0` dans ce cas. On peut être plus précis et exprimer une erreur particulière en renvoyant des autres valeurs négatives.
- une **valeur strictement positive signifie** que l'exécution de la fonction s'est bien déroulée. En général, la fonction renvoie `1` dans ce cas. On peut être plus précis et exprimer une information particulière en renvoyant des autres valeurs strictement positives.

Mécanisme de gestion d'erreurs

Le code d'erreur d'une fonction ayant un prototype standard suit la spécification suivante :

- une **valeur négative ou nulle** renseigne qu'une erreur s'est produite. En général, la fonction renvoie `0` dans ce cas. On peut être plus précis et exprimer une erreur particulière en renvoyant des autres valeurs négatives.
- une **valeur strictement positive signifie** que l'exécution de la fonction s'est bien déroulée. En général, la fonction renvoie `1` dans ce cas. On peut être plus précis et exprimer une information particulière en renvoyant des autres valeurs strictement positives.

Attention : il y a des fonctions de la librairie standard qui ne suivent pas cette spécification.

Mécanisme de gestion d'erreurs

Le code d'erreur d'une fonction ayant un prototype standard suit la spécification suivante :

- une **valeur négative ou nulle** renseigne qu'une erreur s'est produite. En général, la fonction renvoie `0` dans ce cas. On peut être plus précis et exprimer une erreur particulière en renvoyant des autres valeurs négatives.
- une **valeur strictement positive signifie** que l'exécution de la fonction s'est bien déroulée. En général, la fonction renvoie `1` dans ce cas. On peut être plus précis et exprimer une information particulière en renvoyant des autres valeurs strictement positives.

Attention : il y a des fonctions de la librairie standard qui ne suivent pas cette spécification.

De notre côté, nous allons la suivre à la lettre dans les fonctions que nous écrirons.

Mécanisme de gestion d'erreurs

Le code d'erreur d'une fonction ayant un prototype standard suit la spécification suivante :

- une **valeur négative ou nulle** renseigne qu'une erreur s'est produite. En général, la fonction renvoie 0 dans ce cas. On peut être plus précis et exprimer une erreur particulière en renvoyant des autres valeurs négatives.
- une **valeur strictement positive signifie** que l'exécution de la fonction s'est bien déroulée. En général, la fonction renvoie 1 dans ce cas. On peut être plus précis et exprimer une information particulière en renvoyant des autres valeurs strictement positives.

Attention : il y a des fonctions de la librairie standard qui ne suivent pas cette spécification.

De notre côté, nous allons la suivre à la lettre dans les fonctions que nous écrirons.

Remarque : le fonctionnement des codes d'erreurs de chaque fonction écrite doit être spécifié dans sa documentation.

Exemple 1 de gestion d'erreur

Considérons la fonction

```
int division(float x, float y, float *res) {
    if (y == 0)
        return 0;
    *res = x / y;
    return 1;
}
```

Exemple 1 de gestion d'erreur

Considérons la fonction

```
int division(float x, float y, float *res) {
    if (y == 0)
        return 0;
    *res = x / y;
    return 1;
}
```

Les entrées sont les flottants `x` et `y`. La sortie est `res` ; c'est une adresse qui pointera sur le résultat de la division de `x` par `y`.

Exemple 1 de gestion d'erreur

Considérons la fonction

```
int division(float x, float y, float *res) {
    if (y == 0)
        return 0;
    *res = x / y;
    return 1;
}
```

Les entrées sont les flottants `x` et `y`. La sortie est `res` ; c'est une adresse qui pointera sur le résultat de la division de `x` par `y`.

La valeur de retour est un code d'erreur : il vaut `0` lorsque la division ne peut pas être calculée (`y` nul) et vaut `1` sinon.

Exemple 1 de gestion d'erreur

Considérons la fonction

```
int division(float x, float y, float *res) {
    if (y == 0)
        return 0;
    *res = x / y;
    return 1;
}
```

Les entrées sont les flottants `x` et `y`. La sortie est `res` ; c'est une adresse qui pointera sur le résultat de la division de `x` par `y`.

La valeur de retour est un code d'erreur : il vaut `0` lorsque la division ne peut pas être calculée (`y` nul) et vaut `1` sinon.

On remarque que l'on ne modifie pas `*res` lorsque le calcul ne peut pas être réalisé.

Exemple 2 de gestion d'erreur

Considérons la fonction

```
int nb_min_maj(char *chaine, int *res_min, int *res_maj) {
    int i;
    *res_min = 0;
    *res_maj = 0;
    i = 0;
    while (chaine[i] != '\0') {
        if (('a' <= chaine[i]) && (chaine[i] <= 'z'))
            *res_min += 1;
        else if (('A' <= chaine[i]) && (chaine[i] <= 'Z'))
            *res_maj += 1;
        else
            return 0;
        i += 1;
    }
    return i;
}
```

Exemple 2 de gestion d'erreur

Considérons la fonction

```
int nb_min_maj(char *chaine, int *res_min, int *res_maj) {
    int i;
    *res_min = 0;
    *res_maj = 0;
    i = 0;
    while (chaine[i] != '\0') {
        if (('a' <= chaine[i]) && (chaine[i] <= 'z'))
            *res_min += 1;
        else if (('A' <= chaine[i]) && (chaine[i] <= 'Z'))
            *res_maj += 1;
        else
            return 0;
        i += 1;
    }
    return i;
}
```

L'entrée est la chaîne de caractères `chaine`. Les sorties sont `res_min` et `res_maj` ; ces adresses pointeront sur le nombre de minuscules et de majuscules dans `chaine`.

Exemple 2 de gestion d'erreur

Considérons la fonction

```
int nb_min_maj(char *chaine, int *res_min, int *res_maj) {
    int i;
    *res_min = 0;
    *res_maj = 0;
    i = 0;
    while (chaine[i] != '\0') {
        if (('a' <= chaine[i]) && (chaine[i] <= 'z'))
            *res_min += 1;
        else if (('A' <= chaine[i]) && (chaine[i] <= 'Z'))
            *res_maj += 1;
        else
            return 0;
        i += 1;
    }
    return i;
}
```

L'entrée est la chaîne de caractères `chaine`. Les sorties sont `res_min` et `res_maj` ; ces adresses pointeront sur le nombre de minuscules et de majuscules dans `chaine`.

La valeur de retour est un code d'erreur : il vaut `0` si un caractère non alphabétique apparaît dans `chaine` et vaut la longueur de `chaine` sinon.

Fonctions classiques à gestion d'erreurs

La librairie standard du C contient beaucoup de fonctions à gestion d'erreurs. Par exemple :

Fonctions classiques à gestion d'erreurs

La librairie standard du C contient beaucoup de fonctions à gestion d'erreurs. Par exemple :

- `printf` renvoie le nombre de caractères écrits (sans compter `'\0'`);

Fonctions classiques à gestion d'erreurs

La librairie standard du C contient beaucoup de fonctions à gestion d'erreurs. Par exemple :

- `printf` renvoie le nombre de caractères écrits (sans compter `'\0'`);
- `scanf` renvoie le nombre d'affectations réalisées lors de la lecture. La valeur `EOF` est renvoyée si une erreur de lecture a lieu ;

Fonctions classiques à gestion d'erreurs

La librairie standard du C contient beaucoup de fonctions à gestion d'erreurs. Par exemple :

- `printf` renvoie le nombre de caractères écrits (sans compter `'\0'`);
- `scanf` renvoie le nombre d'affectations réalisées lors de la lecture. La valeur `EOF` est renvoyée si une erreur de lecture a lieu ;
- `malloc` renvoie un pointeur vers la zone de la mémoire allouée. Lorsque l'allocation échoue, la valeur `NULL` est renvoyée ;

Fonctions classiques à gestion d'erreurs

La librairie standard du C contient beaucoup de fonctions à gestion d'erreurs. Par exemple :

- `printf` renvoie le nombre de caractères écrits (sans compter `'\0'`);
- `scanf` renvoie le nombre d'affectations réalisées lors de la lecture. La valeur `EOF` est renvoyée si une erreur de lecture a lieu ;
- `malloc` renvoie un pointeur vers la zone de la mémoire allouée. Lorsque l'allocation échoue, la valeur `NULL` est renvoyée ;
- `fopen` renvoie un pointeur sur le fichier ouvert. Lorsque l'ouverture échoue, la valeur `NULL` est renvoyée ;

Fonctions classiques à gestion d'erreurs

La librairie standard du C contient beaucoup de fonctions à gestion d'erreurs. Par exemple :

- `printf` renvoie le nombre de caractères écrits (sans compter `'\0'`);
- `scanf` renvoie le nombre d'affectations réalisées lors de la lecture. La valeur `EOF` est renvoyée si une erreur de lecture a lieu ;
- `malloc` renvoie un pointeur vers la zone de la mémoire allouée. Lorsque l'allocation échoue, la valeur `NULL` est renvoyée ;
- `fopen` renvoie un pointeur sur le fichier ouvert. Lorsque l'ouverture échoue, la valeur `NULL` est renvoyée ;
- `fclose` renvoie `0` si la fermeture du fichier s'est bien déroulée (attention à ce cas particulier). Lorsque la fermeture échoue, la valeur `EOF` est renvoyée.

Fonctions classiques à gestion d'erreurs

La librairie standard du C contient beaucoup de fonctions à gestion d'erreurs. Par exemple :

- `printf` renvoie le nombre de caractères écrits (sans compter `'\0'`);
- `scanf` renvoie le nombre d'affectations réalisées lors de la lecture. La valeur `EOF` est renvoyée si une erreur de lecture a lieu ;
- `malloc` renvoie un pointeur vers la zone de la mémoire allouée. Lorsque l'allocation échoue, la valeur `NULL` est renvoyée ;
- `fopen` renvoie un pointeur sur le fichier ouvert. Lorsque l'ouverture échoue, la valeur `NULL` est renvoyée ;
- `fclose` renvoie `0` si la fermeture du fichier s'est bien déroulée (attention à ce cas particulier). Lorsque la fermeture échoue, la valeur `EOF` est renvoyée.

Remarque : certaines de ces fonctions ont une gestion d'erreurs encore plus sophistiquée et modifient des variables globales comme `errno` (de l'en-tête `errno.h`) pour renseigner précisément sur l'erreur survenue.

Emploi des fonctions à gestion d'erreurs

Les fonctions à gestion d'erreur renvoient des entiers. De ce fait, leur valeur de retour est une expression booléenne.

Emploi des fonctions à gestion d'erreurs

Les fonctions à gestion d'erreur renvoient des entiers. De ce fait, leur valeur de retour est une expression booléenne.

Nous pouvons donc combiner l'appel d'une fonction à gestion d'erreurs avec un test pour traiter l'erreur éventuelle.

Emploi des fonctions à gestion d'erreurs

Les fonctions à gestion d'erreur renvoient des entiers. De ce fait, leur valeur de retour est une expression booléenne.

Nous pouvons donc combiner l'appel d'une fonction à gestion d'erreurs avec un test pour traiter l'erreur éventuelle.

Voici quelques exemples avec les deux fonctions précédentes :

```
if (division(8, a, &b) == 0) {
    /* Traitement de l'erreur lors
     * de la division par zero. */
}
/* Instructions suivantes. */
```

Emploi des fonctions à gestion d'erreurs

Les fonctions à gestion d'erreur renvoient des entiers. De ce fait, leur valeur de retour est une expression booléenne.

Nous pouvons donc combiner l'appel d'une fonction à gestion d'erreurs avec un test pour traiter l'erreur éventuelle.

Voici quelques exemples avec les deux fonctions précédentes :

```
if (division(8, a, &b) == 0) {
    /* Traitement de l'erreur lors
     * de la division par zero. */
}
/* Instructions suivantes. */

if (nb_min_maj("UnDeuxTrois", &a, &b) == 0) {
    /* Traitement de l'erreur lorsque
     * la chaine de caracteres contient
     * des caracteres non alphabetiques. */
}
/* Instructions suivantes. */
```

Interruption de l'exécution

Dans certains cas où une erreur survient, celle-ci peut être irrécupérable. Il faut donc interrompre l'exécution du programme. On utilise pour cela la fonction

```
void exit(int status);
```

de `stdlib.h`, appelée avec l'argument `EXIT_FAILURE` (constante qui vaut `1`).

Interruption de l'exécution

Dans certains cas où une erreur survient, celle-ci peut être irrécupérable. Il faut donc interrompre l'exécution du programme. On utilise pour cela la fonction

```
void exit(int status);
```

de `stdlib.h`, appelée avec l'argument `EXIT_FAILURE` (constante qui vaut `1`).

P.ex.,

```
/* Allocation dynamique. */
tab = (int *) malloc(sizeof(int) * 1024);

/* Verification de son succes. */
if (NULL == tab)
    /* Sur son echec, on interrompt
     * l'execution immediatement. */
    exit(EXIT_FAILURE);
/* Instructions suivantes. */
```

Interruption de l'exécution

Dans certains cas où une erreur survient, celle-ci peut être irrécupérable. Il faut donc interrompre l'exécution du programme. On utilise pour cela la fonction

```
void exit(int status);
```

de `stdlib.h`, appelée avec l'argument `EXIT_FAILURE` (constante qui vaut `1`).

P.ex.,

```
/* Allocation dynamique. */
tab = (int *) malloc(sizeof(int) * 1024);

/* Verification de son succes. */
if (NULL == tab)
    /* Sur son echec, on interrompt
     * l'execution immediatement. */
    exit(EXIT_FAILURE);
/* Instructions suivantes. */
```

Important : on utilisera ce mécanisme d'arrêt principalement dans la fonction `main`. Ailleurs, il faut préférer renvoyer un code d'erreur plutôt que d'interrompre ainsi l'exécution.

Plan

Assertions d'entrée

Lors d'un appel à une fonction, certains arguments peuvent être dans un état incohérent.

Assertions d'entrée

Lors d'un appel à une fonction, certains arguments peuvent être dans un état incohérent.

Au lieu de gérer ces cas de figure par l'usage de codes d'erreur, il est possible de tester l'état des arguments.

Assertions d'entrée

Lors d'un appel à une fonction, certains arguments peuvent être dans un état incohérent.

Au lieu de gérer ces cas de figure par l'usage de codes d'erreur, il est possible de tester l'état des arguments.

Une pré-assertion (ou assertion d'entrée) est un test réalisé dans une fonction pour vérifier si elle est appelée avec des arguments adéquats.

Assertions d'entrée

Lors d'un appel à une fonction, certains arguments peuvent être dans un état incohérent.

Au lieu de gérer ces cas de figure par l'usage de codes d'erreur, il est possible de tester l'état des arguments.

Une pré-assertion (ou assertion d'entrée) est un test réalisé dans une fonction pour vérifier si elle est appelée avec des arguments adéquats.

On utilise la fonction

```
void assert(int a);
```

du fichier d'en-tête `assert.h`.

Assertions d'entrée

Lors d'un appel à une fonction, certains arguments peuvent être dans un état incohérent.

Au lieu de gérer ces cas de figure par l'usage de codes d'erreur, il est possible de tester l'état des arguments.

Une pré-assertion (ou assertion d'entrée) est un test réalisé dans une fonction pour vérifier si elle est appelée avec des arguments adéquats.

On utilise la fonction

```
void assert(int a);
```

du fichier d'en-tête `assert.h`. Elle fonctionne de la manière suivante :

- lorsque l'assertion `a` est **fausse**, l'exécution du programme est **interrompue** et diverses informations utiles sont affichées ;

Assertions d'entrée

Lors d'un appel à une fonction, certains arguments peuvent être dans un état incohérent.

Au lieu de gérer ces cas de figure par l'usage de codes d'erreur, il est possible de tester l'état des arguments.

Une pré-assertion (ou assertion d'entrée) est un test réalisé dans une fonction pour vérifier si elle est appelée avec des arguments adéquats.

On utilise la fonction

```
void assert(int a);
```

du fichier d'en-tête `assert.h`. Elle fonctionne de la manière suivante :

- lorsque l'assertion `a` est **fausse**, l'exécution du programme est **interrompue** et diverses informations utiles sont affichées ;
- lorsque `a` est **vraie**, l'exécution **continue**.

Exemple 1 de fonction avec pré-assertions

Considérons la fonction

```
void afficher_tab(int tab[], int nb) {
    int i;
    assert(tab != NULL);
    assert(nb >= 0);
    for (i = 0 ; i < nb ; ++i)
        printf("%d\n", tab[i]);
}
```

Exemple 1 de fonction avec pré-assertions

Considérons la fonction

```
void afficher_tab(int tab[], int nb) {
    int i;
    assert(tab != NULL);
    assert(nb >= 0);
    for (i = 0 ; i < nb ; ++i)
        printf("%d\n", tab[i]);
}
```

Elle possède deux pré-assertions :

1. la première teste si le tableau `tab` est bien un pointeur valide (différent de `NULL`);

Exemple 1 de fonction avec pré-assertions

Considérons la fonction

```
void afficher_tab(int tab[], int nb) {
    int i;
    assert(tab != NULL);
    assert(nb >= 0);
    for (i = 0 ; i < nb ; ++i)
        printf("%d\n", tab[i]);
}
```

Elle possède deux pré-assertions :

1. la première teste si le tableau `tab` est bien un pointeur valide (différent de `NULL`);
2. la seconde teste si la taille `nb` donnée est bien positive.

Conception de pré-assertions

Il est important de munir ses fonctions de pré-assertions les plus précises et complètes possibles. Quelques règles :

Conception de pré-assertions

Il est important de munir ses fonctions de pré-assertions les plus précises et complètes possibles. Quelques règles :

- la condition testée ne doit dépendre que des arguments d'une fonction (elle ne dépend pas de données apprises à l'exécution) ;

Conception de pré-assertions

Il est important de munir ses fonctions de pré-assertions les plus précises et complètes possibles. Quelques règles :

- la condition testée ne doit dépendre que des arguments d'une fonction (elle ne dépend pas de données apprises à l'exécution);
- la condition testée doit être la plus atomique possible.

```
/* Correct. */
assert(nb >= 0);
assert(nb <= 1024);
```

```
/* Incorrect. */
assert((0<=nb) && (nb<=1024));
```

Conception de pré-assertions

Il est important de munir ses fonctions de pré-assertions les plus précises et complètes possibles. Quelques règles :

- la condition testée ne doit dépendre que des arguments d'une fonction (elle ne dépend pas de données apprises à l'exécution) ;
- la condition testée doit être la plus atomique possible.

```
/* Correct. */
assert(nb >= 0);
assert(nb <= 1024);
```

```
/* Incorrect. */
assert((0<=nb) && (nb<=1024));
```

- Pour les concevoir, il faut imaginer les pires cas possibles à capturer qui peuvent survenir (p.ex., pointeurs nuls, quantités négatives, chaînes de caractères vides, *etc.*).

Conception de pré-assertions

Il est important de munir ses fonctions de pré-assertions les plus précises et complètes possibles. Quelques règles :

- la condition testée ne doit dépendre que des arguments d'une fonction (elle ne dépend pas de données apprises à l'exécution) ;
- la condition testée doit être la plus atomique possible.

```
/* Correct. */
assert(nb >= 0);
assert(nb <= 1024);
```

```
/* Incorrect. */
assert((0<=nb) && (nb<=1024));
```

- Pour les concevoir, il faut imaginer les pires cas possibles à capturer qui peuvent survenir (p.ex., pointeurs nuls, quantités négatives, chaînes de caractères vides, *etc.*).
- Elles sont situées juste après les déclarations de variables dans le corps d'une fonction.

Redondance nécessaire des pré-assertions

Considérons les fonctions

```
int div(int a, int b) {
    assert(b != 0);
    return a / b;
}
```

```
int somme_div(int a, int b) {
    return div(a, b)
        + div(b, a + 1);
}
```

Redondance nécessaire des pré-assertions

Considérons les fonctions

```
int div(int a, int b) {
    assert(b != 0);
    return a / b;
}
```

```
int somme_div(int a, int b) {
    return div(a, b)
        + div(b, a + 1);
}
```

Raisonnement : il n'y a pas de pré-assertion dans `somme_div` mais cela n'est pas grave car les cas problématiques sont capturés par `div` qui contient une pré-assertion.

Redondance nécessaire des pré-assertions

Considérons les fonctions

```
int div(int a, int b) {
    assert(b != 0);
    return a / b;
}
```

```
int somme_div(int a, int b) {
    return div(a, b)
        + div(b, a + 1);
}
```

Raisonnement : il n'y a pas de pré-assertion dans `somme_div` mais cela n'est pas grave car les cas problématiques sont capturés par `div` qui contient une pré-assertion.

Ceci est une **fausse bonne idée** : chaque fonction doit faire ses propres pré-assertions. Toute erreur doit être capturée le plus en amont possible.

Redondance nécessaire des pré-assertions

Considérons les fonctions

```
int div(int a, int b) {
    assert(b != 0);
    return a / b;
}
```

```
int somme_div(int a, int b) {
    return div(a, b)
        + div(b, a + 1);
}
```

Raisonnement : il n'y a pas de pré-assertion dans `somme_div` mais cela n'est pas grave car les cas problématiques sont capturés par `div` qui contient une pré-assertion.

Ceci est une **fausse bonne idée** : chaque fonction doit faire ses propres pré-assertions. Toute erreur doit être capturée le plus en amont possible.

La bonne version de `somme_div` consiste à capturer les mauvaises valeurs possibles de ses arguments de la manière suivante :

```
int somme_div(int a, int b) {
    assert(b != 0);
    assert(a + 1 != 0);
    return div(a, b)
        + div(b, a + 1);
}
```

Exemple 2 de fonction avec pré-assertion

La fonction `nb_min_maj`, à code d'erreur, doit être pourvue de pré-assertions :

```
int nb_min_maj(char *chaine, int *res_min, int *res_maj) {
    int i;

    assert(chaine != NULL);
    assert(res_min != NULL);
    assert(res_maj != NULL);

    /* Suite inchangee. */
}
```

Exemple 2 de fonction avec pré-assertion

La fonction `nb_min_maj`, à code d'erreur, doit être pourvue de pré-assertions :

```
int nb_min_maj(char *chaine, int *res_min, int *res_maj) {
    int i;

    assert(chaine != NULL);
    assert(res_min != NULL);
    assert(res_maj != NULL);

    /* Suite inchangee. */
}
```

On observe que le **mécanisme de gestion d'erreurs** par valeur de retour teste des **comportements** incohérents complexes qui se déroulent à l'**exécution**, tandis que le mécanisme de pré-assertion permet de capturer des erreurs évidentes lisibles directement sur les arguments.

Les pré-assertions pour corriger un programme

Considérons le programme

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

int div(int a, int b) {
    assert(b != 0);

    return a / b;
}

int main() {
    int a;

    a = div(17, 0);
    printf("%d\n", a);

    return 0;
}
```

Les pré-assertions pour corriger un programme

Considérons le programme

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

int div(int a, int b) {
    assert(b != 0);

    return a / b;
}

int main() {
    int a;

    a = div(17, 0);
    printf("%d\n", a);

    return 0;
}
```

La compilation `gcc -ansi -pedantic -Wall Prgm.c` donne l'exécutable `a.out`.

Les pré-assertions pour corriger un programme

Considérons le programme

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

int div(int a, int b) {
    assert(b != 0);

    return a / b;
}

int main() {
    int a;

    a = div(17, 0);
    printf("%d\n", a);

    return 0;
}
```

La compilation `gcc -ansi -pedantic -Wall Prgm.c` donne l'exécutable `a.out`.

Son exécution `./a.out` est interrompue en l.5. Elle produit la réponse

```
a.out: Prgm.c:5: div: Assertion
‘b != 0’ failed.
Aborted (core dumped)
```

On récolte la précieuse information sur le numéro de ligne de la pré-assertion non satisfaite qui produit l'arrêt précipité de l'exécution.