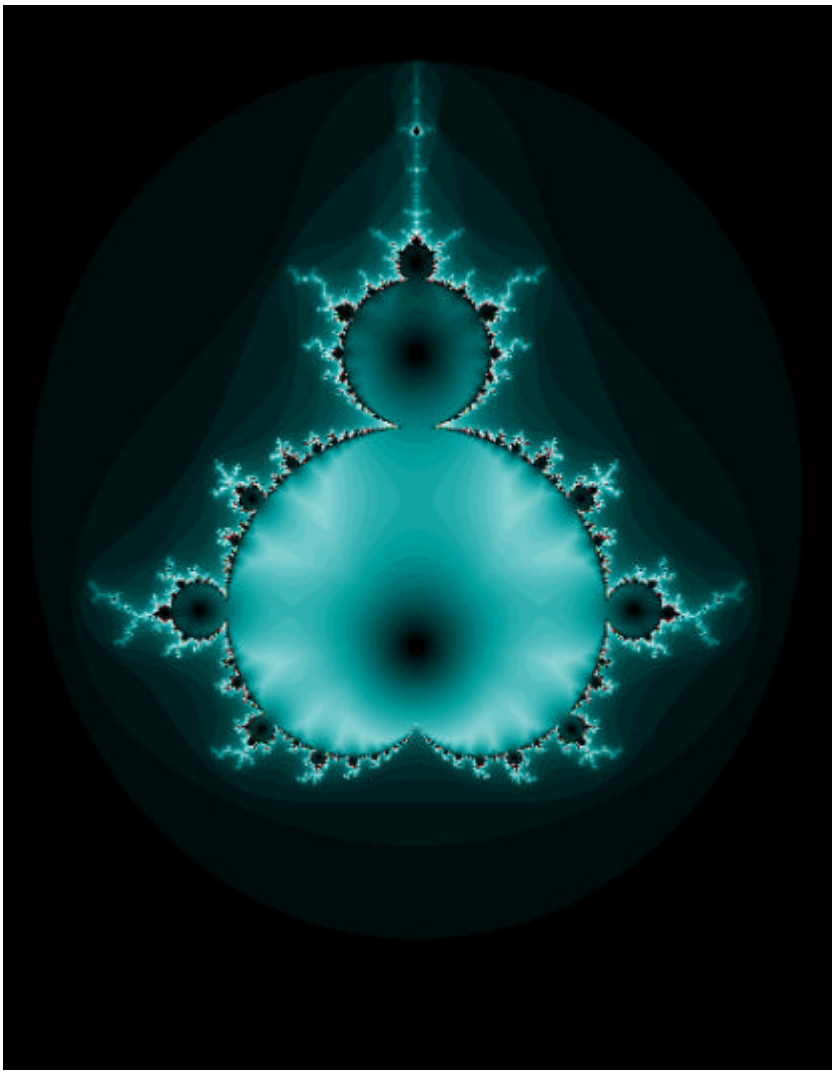
CHAPITRE 5

Le préprocesseur



C++ utilise un préprocesseur avant la phase effective de compilation. Ce préprocesseur permet l'inclusion d'autres fichiers dans le fichier source, la compilation conditionnelle, la génération de macro-instructions, etc... Le préprocesseur se charge également d'éliminer les commentaires du fichier source.

Certains compilateurs C++ sont en réalité des préprocesseurs qui génèrent du code C à partir du code C++. Ce genre de préprocesseur n'est pas notre propos ici: il s'agit d'un préprocesseur standard, ainsi que défini par ANSI C++.

Le préprocesseur n'effectue que des manipulations textuelles, de chaînes de caractères. Il n'a pas de notion de typage, ou de variables. En ce sens, les seules opérations que l'on puisse demander au préprocesseur sont des opérations élémentaires. En C K&R, beaucoup d'opérations (par exemple la définition de constantes) devaient être réalisées à l'aide du préprocesseur. ANSI C, puis C++ a offert la possibilité de corriger ce grave défaut, et il vivement conseillé aux habitués de K&R de se corriger également : c'est à leur bénéfice, même si l'ancienne méthode fonctionne toujours.

5.1 *Commentaires*

C++ autorise deux styles de commentaires:

- les commentaires sur une ligne. Ce type de commentaire est introduit par la chaîne de caractères "//" et se termine à la fin de la ligne courante. Ce type de commentaire n'existe que pour C++, et est illégal en C.
- les commentaires par blocs. Ce type de commentaire est introduit par la chaîne de caractères "/*" et se termine par la chaîne de caractères «*/». Ceci est le style de commentaires classique C.

Ainsi :

```
int       main(int argc, char *argv[])

     {
     int       N = 20; // Ceci est un commentaire C++
     char      *message = "Coucou";
     /*
     Afficher n fois "Coucou" sur l'écran.
     Le présent texte est un commentaire C classique
     */
     for (int i = 0; i < N; i++)
          cout<<message<<endl;

     // Il n'est pas mandatoire de retourner une valeur depuis
     // main(), mais il est considéré comme préférable de
     // le faire.

     return 0;
     }
```

En revanche, la séquence de commentaires suivante est fausse :

```
// Commentaire C++ /* Commentaire C classique
Suite du commentaire C classique
Fin du commentaire C classique */
```

Le début du bloc commentaire C classique n'a pas été vu par le préprocesseur, puisque à l'intérieur d'un commentaire C++. Ajoutons de plus qu'il n'est pas sans autre possible de mettre des commentaires dans des commentaires (*nested comments*) en C conventionnel ou ANSI. En C++, ou en mélangeant judicieusement les styles de commentaires, on peut écrire:

```
/*   Commentaire C, début de bloc:
// Commentaire C++ dans le bloc C
// Autre commentaire C++
```

```
     Fin de commentaire C */
     //   Nouveau commentaire C++ // Commentaire C++ imbriqué sans effet
```

Quelle utilité? Lorsque l'on dépanne un programme, il est souvent intéressant de commenter temporairement tout un bloc de code à l'intérieur d'une procédure pour constater l'effet de l'absence de ce code. Si ce bloc est muni de commentaires C (/* ... */), il sera malaisé de supprimer ce code, puisqu'on ne peut pas imbriquer des commentaires. Si en revanche, ce bloc de code ne contient que des commentaires C++, vous n'aurez aucun problème. En conclusion, utilisez de préférence le style de commentaires C++ («//») lorsque vous programmez.

Lorsque l'on mélange les styles de commentaires, on peut avoir des surprises. Ainsi :

```
int  deuxOuQuatre = 4 //* Commentaire imbriqué */ 2;
;
```

initialise deuxOuQuatre à 4 en C++, et à 2 en C.

Il existe néanmoins un cas où il est préférable de recourir à des commentaires en pur style C, et ce sont les fichiers en-tête (.h) que vous définissez comme pouvant être utilisés avec des compilateurs C et C++ indifféremment.

5.2 *Directives du préprocesseur*

Il existe également un jeu de directives spécifiques au préprocesseur. Ces directives répondent aux règles usuelles définies pour le compilateur C: une directive est signalée par le caractère '#' comme **premier**[1] caractère imprimable d'une ligne, et se termine par un caractère de fin de la ligne.

5.2.1 *Directive #include*

La directive la plus largement utilisée en C++ est certainement le `#include` permettant d'insérer le texte d'un autre fichier. C'est à l'aide de cette directive que l'on va être plus tard à même d'importer les définitions de bibliothèques ou de classe que l'on utilisera.

Il y a deux syntaxes de base pour le fichier que l'on désire inclure :

```
#include      <libfile.h>
     // Inclut un fichier défini dans le système
     //   de développement

#include      "myfile.h"
     //   Inclut un fichier défini par l'utilisateur
```

Sur un système UNIX, les fichiers définis dans le cadre du système se trouvent dans le directoire `/usr/include`, alors que les fichiers définis par l'utilisateur se trouvent dans le directoire de travail. Notons qu'il est possible de définir d'autres directoires. Sous UNIX, la directive de compilation -I permet de spécifier des directoires supplémentaires dans lesquels le compilateur devra chercher les fichiers à inclure. Les environnements plus spécifiques, comme Visual C++ ou Symantec C/C++, permettent de définir ces chemins de recherche alternatifs à l'aide de la notion de projet. Sous UNIX, la directive

```
$ CC -I myIncl -I /usr/X11/include -L /usr/lib/X11 -o myProg
myProg.C
```

compile le programme myProg.C et forme le ficher objet exécutable myProg, en cherchant les fichiers à inclure d'abord dans myIncl, puis dans `/usr/X11/include`, puis finalement dans l'ensemble de directoires prédéfinis (`/usr/include` et le directoire courant). De plus, la directive contient une instruction pour l'éditeur de liens (`ld` sous UNIX) l'informant que les fichiers objet sont à rechercher, en plus de `/usr/lib` et dans le directoire de travail, dans `/usr/lib/X11` également.

Les directives `#include` peuvent figurer à l'intérieur d'un fichier en-tête, c'est-à-dire être imbriquées. Ceci pose un problème, car le même fichier pourrait être inclus plusieurs fois, conduisant à de multiples définitions de mêmes éléments. Dans des environnements de développement intégrés, où toutes les composantes du système de développement sont implémentées dans un seul et même produit, on résoud habituellement ce conflit au niveau du

1. Selon la norme, ce n'est pas mandatoire. En pratique, et en raison des diversités d'interprétation des compilateurs de bas prix disponibles sur le marché, il est préférable de s'en tenir à cette règle.

système de développement: lorsque l'on compile, le système de développement mémorise les fichiers `#include` utilisés, et évite de compiler deux fois le même fichier. Ainsi, dans l'environnement de Symantec, par exemple, il suffit de cocher l'option "`Include each header file only once`" pour voir ce problème résolu. Sous UNIX, il n'existe pas de telles facilités, en partie pour garantir la portabilité du source d'un système à un autre, et on contourne ce problème par l'utilisation judicieuse des directives `#ifdef`, `#ifndef`, `#define`, `#endif` et `#else`.

5.2.2 *Inclusions selon ANSI C++*

ANSI C++ introduit un nouveau style d'include, qui permet d'écrire

```
#include      <cstdlib>
```

en lieu et place de

```
#include      <stdlib.h>
```

Attention ! Cette modification n'est pas aussi innocente qu'elle pourrait en avoir l'air ! Les deux en-tête ne sont pas identiques; la différence se situe au niveau des *namespaces* (voir “Espaces de dénomination (namespace)”, page284), une notion sur laquelle nous reviendrons plus tard. Le fichier inclus dans la première version n'est vraisemblablement pas identique à celui inclus dans la seconde.

Noter aussi :

```
#include      <iostream>
```

en lieu et place de

```
#include      <iostream.h>
```

5.2.3 *Directives de compilation conditionnelle*

`#define` permet de définir le nom qui suit la directive. Ainsi `#define STRINGS_H` définit la chaîne de caractères STRINGS_H comme un nom, alors que `#define aLiteralConstant 1000` définit la chaîne de caractères aLiteralConstant et lui confère la valeur 1000. Le préprocesseur ne fait que substituer ensuite dans le texte la chaîne `aLiteralConstant` par la valeur 1000; cette substitution est purement textuelle.

`#ifdef, #ifndef, #else` permettent de tester la définition ou la non-définition d'un nom, et d'entreprendre une action alternative au besoin. Notons que `#else if` peut aussi s'abréger `#elif`.

`#ifdef` est équivalent à `#if defined`; `#ifndef` est équivalent à `#if !defined`.

On peut ainsi réaliser l'inclusion conditionnelle de fichiers par le mécanisme suivant :

```
#ifndef  THIS_INCLUDE_FILE
#define  THIS_INCLUDE_FILE
/*
Reste du fichier include
*/
#endif
```

A la premiere inclusion du fichier, THIS_INCLUDE_FILE n'est pas défini, la condition de la première ligne est fausse, et la seconde ligne est exécutée, définissant ainsi THIS_INCLUDE_FILE. Ensuite, le reste du fichier inclus est traité normalement. A la deuxième inclusion, THIS_INCLUDE_FILE est défini, et le fichier est ignoré jusqu'à l'occurence du #endif correspondant, ce qui correspond à l'effet recherché. Précisons néanmoins que cette technique nécessite de bien synchroniser la définition de ces chaînes de caractères, par un algorithme approprié, surtout si le groupe de développement est formé d'un grand nombre de programmeurs.

5.2.4 *Macro-instructions*

En C (conventionnel et ANSI) `#define` est beaucoup utilisé pour définir des macros[1]. Un exemple typique, cité dans la quasi-totalité des manuels, est la macro générant la valeur maximum de deux nombres :

```
#define  MAX(a, b) ((a) > (b) ? (a) : (b))
```

Le nombre d'inconvénients de cette formulation est si grand que nous renonçons à être exhaustifs dans le cas présent. Même si on ne tient pas compte de l'écriture lourde et cryptique, encombrée de paranthèses, il est toujours possible de tomber dans l'un des pièges suivants :

```
int  a = 1, b = 0;

MAX(a++, b + 10);       // a est incrémenté une fois !
MAX(a++, b);            // a est incrémenté deux fois.
MAX(a, "Coucou");       // Comparaison d'entiers et de pointeurs.
```

Le dernier exemple génère au minimum un message d'avertissement de la part de la majorité des compilateurs, mais ce message concernera le code généré par le préprocesseur lors de l'expansion de la macro, et non le code que vous avez généré vous-même. Dans certains cas, cela risque de poser des problèmes de localisation de l'erreur. C++ propose des outils autrement puissants pour ce genre de problèmes, comme nous le verrons plus en détail ultérieurement. Ainsi, la ligne suivante

1. Les macros sont une survivance particulièrement inopportune de C. En tant que telles, il faut essayer par tous les moyens de les éviter. C++ offre en général des possibilités nettement plus propres et plus efficaces pour effectuer les opérations attendues de la part d'une macro. Un usage abusif de macros en C++ dénote une connaissance insuffisante du langage C++.

```
inline    int       MAX(int a, int b) {return a > b ? a : b}
```

remplit le même usage que le #define çi-dessus, mais ne possède aucun de ses inconvénients. Nous verrons d'autre part qu'il est possible de réaliser encore mieux au moyen de la construction C++ template.

5.2.5 *#undef*

Toute variable ou macro ayant été définie avec #define peut être détruite par la directive #undef.

```
#define   NOM_DE_MACRO

...

#undef    NOM_DE_MACRO
```

5.3 *Identificateurs prédéfinis*

5.3.1 `__cplusplus`

Cet identificateur est défini lorsque le compilateur traitant le code est un compilateur C++. Cet identificateur est particulièrement utile pour ceux qui désirent définir un fichier de définition (.h) pouvant être utilisable aussi bien en C qu'en C++.

```
#ifdef __cplusplus
extern   C    {
#endif

/* reste du fichier de définition */

#ifdef __cplusplus
     }        /* Fermer l'accolade ouverte plus haut */
#endif
```

5.3.2 *__LINE__*

Cet identificateur a pour valeur le numéro de la ligne courante. Une application intéressante est le traçage d'évènements au sein d'un programme complexe. Des évènements imprévus par le programmeur (comme l'inexistence d'un fichier qui devait se trouver là, par exemple) peuvent être tracés sur un fichier de traçage, fournissant ainsi au groupe de développement un moyen de diagnostiquer des pannes apparaissant chez un client.

5.3.3 *__FILE__*

Cet identificateur a pour valeur le nom du fichier source courant. On l'utilisera volontiers en conjonction avec `__LINE__` pour signaler des évènements anormaux compromettant l'exécution normale du programme, comme dans l'exemple suivant :

```
...
ifstream ressourceFile;

if (!ressourceFile.open("ressource.dat"))
     // Fichier ressource n'a pas pu être ouvert
     // continuation du programme impossible
     {
     cerr<<"Impossible d'ouvrir fichier ressource a la ligne"<<
               __LINE__<<" du fichier "<<"__FILE__"<<endl;
          // Interruption abortive du programme
          // Voir <stdlib.h>
     abort();
     }
...
```

5.3.4 *__DATE__ et __TIME__*

Ces deux identificateurs contiennent la date et l'heure, respectivement. Attention : il s'agit bien évidemment de la date et l'heure du moment du passage du préprocesseur. Il serait vain de vouloir utiliser ces deux identificateurs pour connaître l'heure au moment de l'exécution du programme !

Une utilisation fréquente de ces variables consiste à documenter les compilations effectuées dans de grands projets, où la recompilation du projet entier peut prendre plusieurs heures (voire plusieurs jours dans certains cas célèbres !).

5.3.5 *#line*

Cette directive permet de modifier les variables __LINE__ et __FILE__.

```
#line     NOUVEAU_NUMERO_DE_LIGNE     "NOUVEAU_NOM_DE_FICHIER"
```

La mention d'un nouveau nom de fichier est optionnelle.

5.3.6 *#error*

Cette directive permet d'émettre un message d'erreur sur la sortie erreur standard lors de la précompilation. On utilise parfois cette directive pour avertir l'utilisateur d'une option manquante lors de la compilation.

```
#ifdef   __DEBUG
#error   Il faut compiler avec l'option -g !
#endif
```

5.3.7 *#pragma*

Cette directive provoque un comportement dépendant de l'implémentation lorsque la séquence de symboles suivant la directive est reconnue par ladite implémentation. On utilise volontiers des pragmas pour provoquer des comportements particuliers à une machine donnée. Les pragmas sont spécifiques à un type de machine déterminée: un pragma non reconnu sera ignoré.

```
#pragma NOOPTIMIZE
```

5.4 *Test*

1. Comment pourrait-on implémenter dans un programme un message documentant une erreur en cours d'exécution de manière significative pour le programmeur, comme par exemple :

   Erreur fatale dans le fichier source "fichiers.cpp", à la ligne 28 : "Fichier inexistant".

   On aimerait disposer d'un mécanisme facile à utiliser pour accéder à ce service, de manière à ce que les programmeurs qui veulent documenter les erreurs en cours d'exécution puissent le faire de façon uniforme et facilement.
2. Définissez un en-tête de fichier personnalisé qui indique automatiquement :

   * Le nom du fichier source

   * La date de la dernière compilation

   * L'heure de la dernière compilation