Précompilation

INTRODUCTION

- Intervient avant la compilation

  Actions possibles :
  - Inclusion d'autres fichiers dans le fichier à compiler.
  - Définition des constantes symboliques et des macros.
  - Compilation conditionnelle de code de programme
  - Exécution conditionnelle des directives du précompilateur.

- Directives de précompilation :

  - elles commencent par un #,
  - elles ne sont pas des instructions C++,
  - elles sont traitées avant la compilation.

Directives de précompilation #include

- Provoque l'insertion d'une copie d'un fichier spécifié à la place de la directive.

- 2 formes possibles :

  #include <nomfichier>

  Recherche du fichier dans des répertoires prédéfinis.

  Ex. : #include <iostream.h>
  #include <iomanip.h>

  #include "nomfichier"

  Recherche du fichier dans le même répertoire que le fichier compilé, puis, le cas échéant, poursuit la recherche de la même façon que pour les fichiers entourés de crochets.

  Ce sont des fichiers d'en-tête définis par le programmeur renfermant les classes, les structures, les unions, les énumérations et les prototypes de fonctions.

Directives de précompilation #define : les constantes symboliques

- Permet de créer des constantes représentées comme des symboles.

- Format général :

#define identificateur texte_de_remplacement

Ex. : #define PI 3.14159

Toutes les occurrences suivantes de identificateur sont remplacées par texte_de_remplacement avant la compilation.

Erreur : #define PI = 3.14159

- Peut être utilisé seulement dans le fichier dans lequel ces constantes sont définies.
- En C++, on préfère utiliser des variables **const** car elles ont un type de donnée spécifique et sont visibles par leur nom pour un débogueur.
- Pour des constantes symboliques, seul le texte de remplacement est visible au débogueur.

Directives de précompilation #define : macros

- Utilisé en C ou avec du code hérité du C.
- En C++, les macros ont été remplacées par les modèles et les fonctions inline.
- 2 cas :

(A) Macros sans arguments

Traitées comme des constantes symboliques

Ex. : #define AFFICHE() (cout << endl << str << endl;)

(B) Macros avec arguments

- Les arguments sont substitués dans le texte de remplacement.
- La macro est développée i.e. le texte de remplacement se substitue, dans le programme, à l'identificateur de macro et à sa liste d'arguments.

Note : - aucune vérification de type de donnée n'est effectuée sur les arguments de macro.
- une macro n'est utilisée que dans le cadre d'une substitution de texte.

Ex. : #define AIRE_DU_CERCLE(x) (PI * (x) * (x))

Chaque fois que AIRE_DU_CERCLE(x) apparaît dans le fichier,
- la valeur de x est substituée à x dans le texte de remplacement,
- la constante symbolique PI est remplacée par sa valeur.

Si nous avons dans le programme,

```
aire = AIRE_DU_CERCLE( 4 );
```

cela est développé en

```
aire = ( 3.14159 * ( 4 ) * ( 4 ));
```

Comme l'expression n'est constituée que de constantes, la valeur de l'expression est évaluée lors de la compilation et le résultat est affecté à aire au moment de l'exécution.

Note : Ne pas oublier les parenthèses, cela impose l'ordre d'évaluation correct.

De plus, l'instruction

```
aire = AIRE_DU_CERCLE( c + 2 );
```

est développée en

```
aire = ( 3.14159 * ( c + 2 ) * ( c + 2 ));
```

La macro AIRE_DU_CERCLE pourrait être définie comme une fonction :

```
double aireCercle( double x )
        {       return 3.14159 * x * x;         }
```

cela impliquerait une surcharge de travail induite par l'appel de fonction.

- Désavantage de la macro : augmente la taille du programme.
- Les fonctions inline sont préférables aux macros parce qu'elles assurent les services de vérification de type des fonctions.

Exemple :

```
#define AIRE_DU_RECTANGLE(x, y) (( x ) * ( y ))
```

aireRect = AIRE_DU_RECTANGLE(a + 4, b + 7); est développée en :

aireRect = ((a + 4) * (b + 7));

Note : (i) Le texte de remplacement pour une macro ou une constante symbolique est normalement constitué de tout le texte qui suit l'identificateur dans la directive #define sur la ligne.

Si le texte de remplacement est plus long que le reste de la ligne, une barre oblique inverse (\ ) placée à la fin de la ligne indique que ce texte continue sur la ligne suivante.

(ii) On peut supprimer les constantes symboliques et les macros à l'aide de la directive de précompilation #undef.

Portée d'une constante symbolique ou d'une macro :

De sa définition jusqu'à la suppression de la définition par #undef ou jusqu'à la fin du fichier.

(iii) Une fois sa définition supprimée, un nom peut être redéfini par un #define.

Compilation conditionnelle

- Permet de contrôler l'exécution des directives de précompilation et la compilation du code de programme.

- Cela évalue une expression entière constante qui détermine si le code doit être compilé.

- Ex. :
```
#if !defined(compiler)
        #define compiler 0
#endif
```

Si compiler est défini, !defined(compiler) est 0 et la directive #define est omise. Autrement, defined(compiler) s'évalue à 0 et compiler est défini.

- Les directives #ifdef et #ifndef sont des raccourcis pour #if defined(nom) et #if !defined(nom).

- Utilités :
  - éviter d'inclure le même fichier plusieurs fois,
  - aide au débogage permettant de mettre des portions de code en commentaire pour éviter que ce code soit compilé.

```
#if 0
        code omis à la compilation
#endif
```

Note : En cours de route, si la valeur 0 est remplacée par 1, alors la compilation est permise.

  - Afficher les valeurs des variables et confirmer le flot de contrôle.

```
#ifdef DEBOGAGE
        cerr    << " Variable x = "
```

```
        << x << endl;
#endif
```

Nous pouvons forcer la compilation de l'instruction cerr dans le programme si la constante symbolique DEBOGAGE a été définie, par un #define DEBOGAGE avant la directive précédente.

Une fois le débogage terminé, la directive #define est retirée et l'instruction cerr est ignorée.

Directives de précompilation #error et #pragma

La directive #error permet d'afficher un message.

```
#error message
```

Ex. :

```
#if !defined(__cplusplus)
#error Un compilateur C++ est requis.
#endif
```

Le message est affiché. La précompilation s'arrête et le programme n'est pas compilé.

La directive #pragma provoque une action définie au niveau de l'implantation. Plusieurs possibilités s'offrent à nous; référez-vous à la documentation disponible avec votre compilateur C++.

Ex. : Permet d'afficher le nom du fichier à compiler, la date et le moment où ce fichier a été modifié pour la dernière fois.

```
#pragma message("Compiling " __FILE__ )
#pragma message("Last modified on " __TIMESTAMP__ )
```

Opérateur de précompilation

Cet opérateur provoque la conversion d'une série de caractères du texte de remplacement en une chaîne de caractères entourée de guillemets.

Ex. : #define BONJOUR(x) cout << " Bonjour, " #x << endl

Si BONJOUR(Jean); apparaît dans un programme, il est développé en :

```
cout << " Bonjour, " "Jean" << endl;
```

Ces deux chaînes seront concaténées.

Opérateur de précompilation

Cet opérateur provoque la conversion d'une série de caractères du texte de remplacement par une série de caractères passée en paramètre.

Ex. :

```
#define AFFICHE(n) (cout << endl << s##n << endl)

using namespace std;

void main()
{
	string s1, s2("OUF"), s3("p");

	AFFICHE(1);
	AFFICHE(2);
	AFFICHE(3);
}
```

Numéros de lignes #line

Cela provoque la renumérotation des lignes de code source qui suivent la valeur constante entière spécifiée.

Ex. : `#line 100`

Démarre la numérotation à 100 à partir de la prochaine ligne de code source.

On peut aussi inclure un nom de fichier dans la directive.

```
#line 100 "FichierA.cpp"
```

En plus, le nom du fichier "FichierA.cpp" apparaîtra dans tout message du compilateur.

Note : Les numéros de lignes n'apparaissent pas dans le fichier source.

Constantes symboliques prédéfinies

LINE	Numéro de ligne en cours du code source (une constante entière)
FILE	Nom présumé du fichier source (une chaîne de caractères)
DATE	Date de compilation du fichier source (une chaîne de caractères de la forme "Mmm jj aaaa" telle que "Jan 19 1998").

TIME Heure de compilation du fichier source (une chaîne littérale de la forme "hh:mm:ss").

STDC Constante entière 1. Ceci a pour but d'indiquer que l'implantation est compatible avec la norme ANSI.

Ex. : #pragma message("Compiling " __DATE__)
#pragma message("Last modified on " __TIME__)

Note : Ces 5 identificateurs et l'identificateur defined ne peuvent être utilisés dans des directives #define ou #undef.

Assertions

La macro assert, définie dans le fichier en-tête <assert.h>, teste la valeur d'une expression.

Si la valeur de l'expression vaut 0 alors assert affiche un message d'erreur et appelle la fonction abort de la bibliothèque d'utilitaires généraux <stdlib.h> pour terminer l'exécution du programme.

Ex. : assert(x <= 10);

Si x > 10 alors un message d'erreur contenant le numéro de ligne et le nom du fichier est affiché et le programme se termine.

Note : Si la constante symbolique NDEBUG est définie,

#define NDEBUG

les assertions qui suivent seront ignorées. Cela évite d'enlever les assertions après la mise au point.

oooooooooooooooooooooooooooooooooooooooooooo