Un code source en langage C, lors de sa compilation, est d'abord lu par un *préprocesseur* : c'est un logiciel qui va lire le code source et appliquer des modifications dessus, avant qu'il soit effectivement compilé.

Le C++ étant une extension du C (tout code fonctionnant en C fonctionnera en C++, moyennant quelques retouches éventuelles), il fait lui aussi intervenir le préprocesseur. Il est néanmoins préférable d'utiliser les fonctionnalités du C++ à la place lorsque cela est possible.

Ce préprocesseur effectue deux tâches : il supprime les commentaires du code source, et il exécute les lignes qui commencent par un dièse « `#` » (puis les retire du code) comme les `#include` que nous recopions à chaque fois :

```
#include <iostream>
```

Nous allons dans ce cours vous détailler le fonctionnement de certaines de ces directives. Cela est surtout destiné à vous permettre de comprendre d'autres programmes en C et en C++, et à mieux connaître les possibilités et mécanismes des langages. Vous ne devriez pas en avoir énormément besoin dans la suite des exercices.

Inclusion de fichiers

Une directive `#include` peut s'écrire de deux manières : avec des guillemets ou avec des chevrons. Ainsi, dans certains exercices, on a écrit :

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

Dans les deux cas, le préprocesseur va chercher le fichier portant le nom indiqué (celui-ci peut d'ailleurs être un chemin, comme `entetes/declarations.h`), puis insérer l'intégralité de son contenu à l'emplacement de la directive.

Le choix entre les guillemets et les chevrons se fait en fonction de l'emplacement du fichier à inclure. La convention est la suivante : s'il s'agit d'un fichier d'en-tête installé dans l'environnement de compilation, il faut écrire des chevrons `<>` ; sinon, si c'est un fichier lié au projet en cours de développement, il faut écrire des guillemets `""`. Avec les guillemets, le préprocesseur va commencer par chercher le fichier à inclure dans le même dossier que le fichier qu'il est en train de traiter. Dans les deux cas, le fichier est ensuite recherché dans des emplacements prévus pour stocker des fichiers d'en-tête (des répertoires s'appelant généralement `include`). En pratique, on pourrait donc ne jamais utiliser de chevrons et toujours employer des guillemets (mais c'est moins informatif).

Sur notre plateforme, vous n'aurez jamais d'autre fichier à inclure que ceux de la bibliothèque standard de votre langage ou quelques fichiers liés aux exercices.

Notez qu'une fois le fichier inclus, son contenu est traité par le préprocesseur (rassurez-vous, les cycles d'inclusions seront détectés).

Définition de symboles et macros

La directive `#define` permet de définir des symboles et des macros. Un symbole est simplement un nom, que l'on peut écrire dans le code source, et qui sera remplacé par un code spécifié dans la définition. Par exemple :

```
#define NB_MAX_JOUEURS (200 * 1000)
#define INFINI (1000 * 1000 * 1000)
```

Les parenthèses peuvent s'avérer nécessaires selon la position de la macro. Par exemple, cette instruction :

```
double frequenceAbsents = (double)joueursAbsents / NB_MAX_JOUEURS;
```

sera transformée en :

```
double frequenceAbsents = (double)joueursAbsents / (200 * 1000);
```

Sans les parenthèses, on obtiendrait un résultat différent !

Une macro fonctionne de la même façon, mais accepte en plus des paramètres. C'est le cas de la macro *repeat* que nous vous avons suggéré d'utiliser au début du cours :

```
#define repeat(nb) for (int _loop = 1, _max = (nb); _loop <= _max; _loop++)
```

Avec cette définition, lorsque l'on écrit `repeat()` dans la suite du code avec une valeur pour *nb* entre les parenthèses, alors le `repeat()` va être remplacé par le code indiqué dans la définition, `nb` y étant remplacé par le paramètre fourni. Voici un exemple farfelu illustrant le fonctionnement :

```
int saisie;
cin >> saisie;
repeat (saisie)
{
   cin >> saisie;
   cout << saisie << endl;
}
```

Il s'agit d'une implémentation malpropre de l'algorithme suivant :

```
nbEntiers <- lireEntier()
Répéter nbEntiers fois
   nb <- lireEntier()
   Afficher nb
```

Après le passage du préprocesseur, ce code est donc transformé en :

```
int saisie;
cin >> saisie;
for (int _loop = 1; _max = (saisie); _loop <= _max; _loop++)
{
   cin >> saisie;
   cout << saisie << endl;
}
```

La boucle crée donc deux variables *_loop* et *_max*. Cette dernière est initialisée à la valeur donnée en paramètre, et *_loop* va de 1 jusqu'à ce *_max*.

Nous avons mis des blancs devant les noms de variable pour éviter qu'ils correspondent à des identifiants que vous auriez créés vous-même. D'autre part, nous avons utilisé *_max* pour fixer le nombre de répétitions au départ, dans le cas où la valeur donnée serait modifiée, ou même difficile à calculer. Notez en outre qu'il est troublant de modifier le nombre de répétitions dans la boucle comme on le fait là : c'est une pratique à éviter.

Enfin, les parenthèses autour de *nb* contribuent également à augmenter la cohérence de la macro. Dans le code du `for`, *nb* est l'opérande de droite d'une affectation. L'affectation est l'opération ayant la plus basse priorité, sauf par rapport à un seul opérateur : la virgule `,` ! Celle-ci permet d'écrire deux expressions indépendantes dans une seule instruction. Vous pouvez étudier sur les lignes ci-dessous pour répondre à vos éventuelles questions (on fait intervenir un type structuré *Ouvrier*) :

```
repeat (10, nbFois)
{
   cout << "Un affichage exécuté " << nbFois << " fois, car (a, b) retourne b." << e
}

Ouvrier ouvriers[NB_MAX_OUVRIERS];
int iOuvrier;
Ouvrier* pOuvrier;
for (iOuvrier = 0, pOuvrier = ouvriers; iOuvrier < nbOuvriers; iOuvrier++, pOuvrier+
{
   cout << "Ouvrier n°" << iOuvrier << " : " << pOuvrier->nom << endl;
}
```

Les macros et symboles sont un outil assez fourre-tout ; on peut par exemple en utiliser pour remplacer les opérateurs logiques par des noms et obtenir un code plus verbeux :

```
#define and &&
#define or ||
#define not !
```

Cela peut être très intéressant pour développer une syntaxe appuyée sur les fonctionnalités du C et du C++. Mais tant que les normes actuelles vous conviennent, nous vous recommandons bien sûr vivement de vous y conformer, afin que votre code puisse facilement être interprété par un autre programmeur !

Les macros peuvent servir à effectuer certains calculs, avec l'avantage qu'elles sont indépendantes du type. Voici quelques exemples :

```
#define abs(nb) ((nb) < 0 ? -(nb) : (nb))
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
```

Nous déclarons trois macros, avec un ou deux paramètres. Notez qu'il ne faut surtout pas mettre d'espace entre le nom de la macro et les parenthèses qui définissent ses paramètres, comme dans :

```
#define macro_erronee (arg) (arg * 2)
```

Ici, on déclarera alors un symbole *macro_erronee* correspondant au code `(arg) (arg * 2)`.

Pour finir, les définitions de symboles et macros peuvent être annulées avec la directive `#undef` :

```
#undef repeat
#undef macro_erronee
```

De manière générale, les symboles et les macros sont déconseillées en C++, pour leur effet fourre-tout. Les fonctionnalités du langage doivent être utilisées à la place. En C, les constantes dont la valeur est connue à la compilation ne peuvent pas être utilisées à certains endroits du programme (comme un cas de sélection : le `case` d'un `switch`), et une fonction ne peut prendre que des paramètres d'un type donné. En C++, les constantes fixées à la compilation peuvent être utilisées partout, et la surcharge et les patrons de fonctions (mot-clé `template`) permettent de généraliser les fonctions à plusieurs types.

Conditions

Il est possible de conditionner l'inclusion d'une partie du code dans le programme : on parle de compilation conditionnelle. La condition peut s'appuyer sur les symboles définis antérieurement dans

le code. Cela permet donc d'inclure des lignes de code éparpillées en changeant une ligne. Par exemple, si l'on inclut des instructions de débogage dans son programme :

```
string clients[nbClients];
for (int iClient = 0; iClient < nbClients; iClient++)
{
   getline(cin, clients[iClient]);
   #ifdef DEBUG
      cout << "Tricherie : " << clients[iClient] << endl;
   #endif
}
int clientMecontent;
cin >> clientMecontent;
#ifdef DEBUG
   cout << "Tricherie : " << clientMecontent << " ; " clients[clientMecontent] << en
#endif
```

(Du débogage, ou appelez ça comme vous voulez !) On pourra facilement décider de les inclure avec la ligne suivante en haut du programme :

```
#define DEBUG
```

La commande `#ifdef` permet ainsi de ne pas inclure le code allant jusqu'au `#endif` correspondant, selon que le symbole indiqué est défini au moment où on y arrive. Cela est notamment très utilisé pour éviter d'inclure un fichier d'en-tête deux fois ; par exemple, dans *robot.h*, on peut faire ainsi :

```
#ifndef ROBOT_H
#define ROBOT_H

// Contenu…

#endif
```

La première fois que le fichier est inclus, le symbole *ROBOT_H* n'existe pas, et le contenu du fichier est alors inclus. Le symbole est dès lors défini. Si jamais le fichier est réinclus, alors *ROBOT_H* est défini et l'inclusion n'a pas d'effet. Par exemple, de nombreux fichiers d'en-tête de la bibliothèque standard du C et du C++ incluent les mêmes fichiers ; il faut donc éviter les collisions de déclarations si l'on inclut ces fichiers.

Il existe ainsi `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else` et `#endif`. Avec `#if` et `#elif`, on peut mettre une condition ; on peut y faire intervenir le mot-clé `defined` :

```
#if (defined DEBUG) && (VERBOSE > 2)
   // Afficher des informations de niveau supérieur à 3
#endif
```

Ces instructions peuvent s'imbriquer. Avec `#if 0`, on peut ainsi facilement inhiber une partie du code, même si elle contient des commentaires (car le 0 représente le « faux ») :

```
#if 0
   /* Un commentaire qui se croit malin */
   // Code
#endif
```

Conclusion

Dans la suite des cours, nous n'utiliserons au final que `#include` pour inclure des déclarations, et `#define` pour définir des symboles, comme nous l'avons déjà fait jusqu'à présent.