Jusqu'à présent, les programmes que nous réalisons sont de toute petite dimension. Toutefois, lorsque nous construisons un véritable projet, il est assez fréquent d'avoir plusieurs milliers de ligne de code. Il n'est pas envisageable, dans ce cas là, que toutes ces lignes soient écrites sur un seul et unique fichier. C'est trop difficile à maintenir et lorsque que nous devons modifier une seule ligne, il est nécessaire de tout recompiler, ce qui représente un temps considérable vu le petit changement réalisé.

Par ailleurs, pour faciliter l'élaboration d'un projet, il est préférable de le découper et de le structurer en plusieurs fonctions. Cette approche a également le mérite de favoriser le travail en équipe. Toutefois, pour que cela soit vraiment efficace, il faut découper le projet en plusieurs fichiers, pour que chacun s'occupe de sa propre tâche.

Pour bien comprendre les mécanismes en jeu, nous allons revenir sur la notion de compilation. Rappelons que la compilation consiste à réaliser une traduction d'un langage de programmation (ici le C++) vers le langage binaire, le seul qui soit compréhensible par le microprocesseur. Cette compilation s'effectue en trois phases :

1. La première phase consiste à effectuer des changements de texte dans le code source, comme par exemple, pour les constantes et les directives de compilations (***« # »***). ***C'est le préprocesseur qui effectue cette opération.***
2. Une fois que le texte définitif est en place, le code source est ensuite analysé pour être traduit en binaire. Cette traduction est alors placée dans un fichier séparé, appelé fichier ***« objet »***. ***C'est la phase de compilation.***
3. Pour finir, la troisième phase rassemble les morceaux (les différents fichiers objets) afin d'obtenir un seul et unique fichier qui sera l'exécutable. ***C'est la phase d'édition de lien.***

## Compilation et édition de lien

Nous allons élaborer un projet avec plusieurs fichiers sources. Chacun de ces fichiers disposera d'une petite fonction. Il est vrai que normalement, nous n'aurions pas eu besoin d'avoir autant de fichiers. Nous allons juste procéder de cette façon pour bien maîtriser les différents concepts mis en jeu.

Principal.cpp | impaire.cpp | paire.cpp

```
//------------------------------
bool impaire (int x)
{
   return x % 2;
}
//------------------------------
```

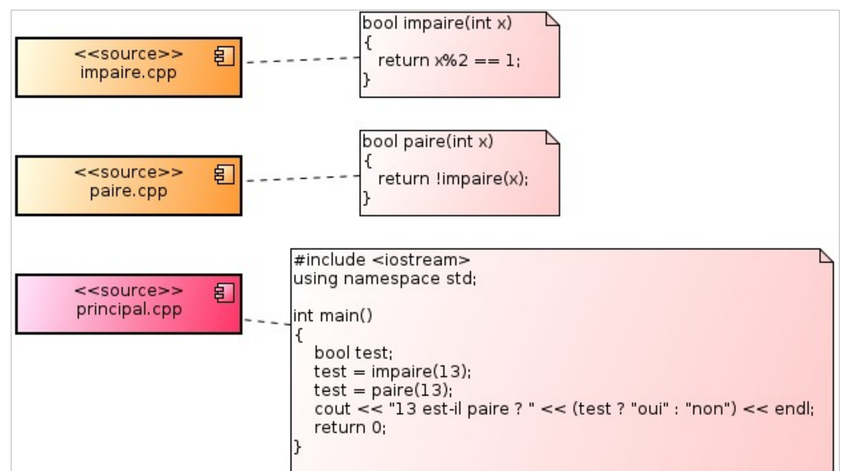
Principal.cpp | impaire.cpp | paire.cpp

```
//------------------------------
bool paire(int x)
{
   return !impaire(x);
}
//------------------------------
```

Principal.cpp | impaire.cpp | paire.cpp

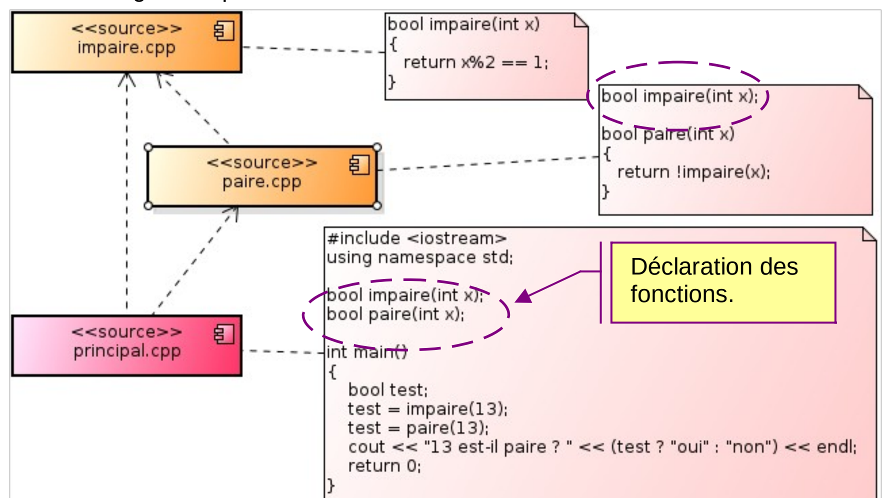
```
//------------------------------
int main()
{
   bool test;
   test = impaire(13);
   test = paire(13);
   return 0;
}
//------------------------------
```

1. Le premier fichier définie la fonction ***impaire()*** et sera appelé ***« impaire.cpp »***.
2. Le deuxième fichier définie de la fonction ***paire()*** qui fait appel à la fonction impaire. Il sera appelé ***« paire.cpp »***.
3. Enfin, la fonction ***main()*** sera définie dans le fichier ***« principal.cpp »***. Cette fonction fera appel respectivement à la fonction impaire et à la fonction paire.




Lorsque nous lançons la compilation, nous obtenons une erreur au moment de l'évocation de la fonction impaire. Une fonction doit toujours être connue, soit par une définition au préalable, soit par une simple déclaration. Comme les fonctions ne sont pas définies dans le même fichier, nous sommes obligés de prendre la deuxième solution. D'ailleurs, la déclaration des fonctions a justement été créée pour répondre à ce genre de problème.

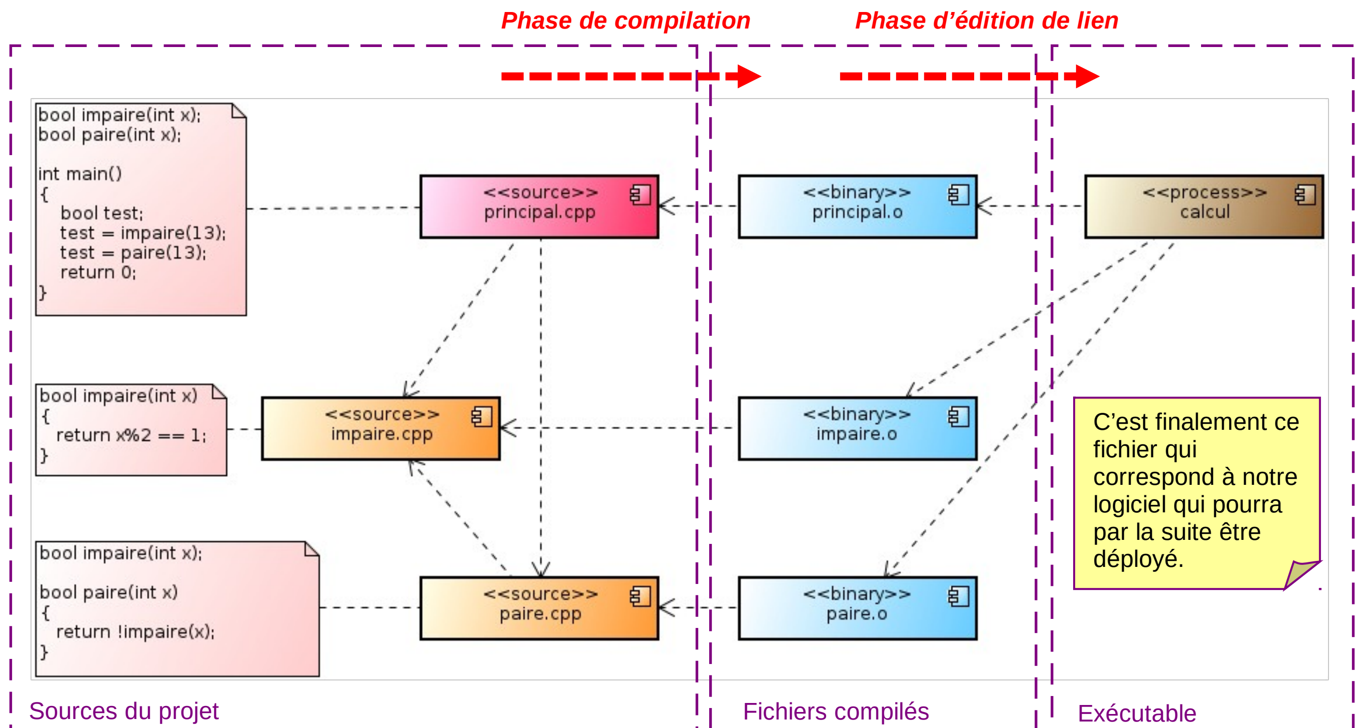
Partout où vous avez besoin d'utiliser des fonctions, il sera donc nécessaire de faire une déclaration au préalable. C'est donc le cas pour les fichiers ***principal.cpp*** et ***paire.cpp***.



Que se passe-t-il réellement lorsque nous effectuons la compilation ?

En introduction, nous avons évoqués trois phases de compilation. La première n'est pas utilisée puisqu'il n'y a aucun texte à changer. Il reste donc la compilation qui opère la traduction et produit les fichiers binaires correspondant : ***« principal.o »***, ***« impaire.o »***, ***« paire.o »***.

A partir de là, c'est ensuite l'éditeur de lien qui prend le relais et qui, comme son nom l'indique, assure la liaison entre les différents modules compilés pour créer un seul fichier qui deviendra l'exécutable de notre application, savoir : ***« calcul »***.

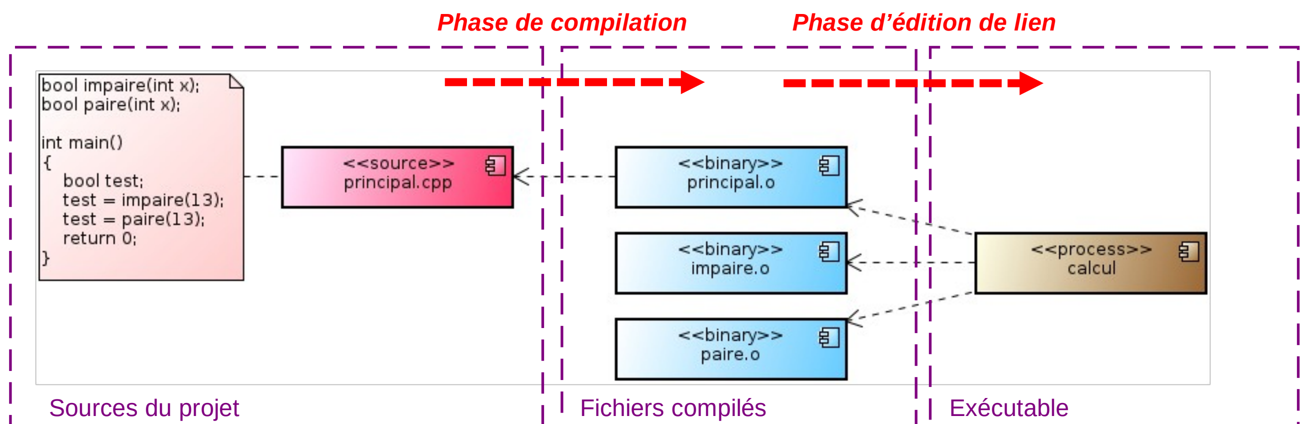



**Make :**
Quelque soit le système de développement intégré, lorsque nous demandons une compilation, le système fait appel au 'make' qui est un outil très puissant qui s'occupe de toutes les phases de compilation. Cet outil ***make*** n'effectue la compilation que sur les fichiers sources qui ont subits une modification. De même, l'édition de lien ne sera lancée que si, au moins, un fichier compilé a subi une modification. Avec ce principe le temps de compilation devient extrêmement réduit et c'est un des atouts de la gestion de projets multi fichiers.

**Construire – Built :**
Vous pouvez toutefois imposer de tout reconstruire même si certains fichiers sources n'ont pas été modifiés. Dans ce cas là, tout se passe comme si toutes les phases de compilation étaient faites pour la première fois.

## Travailler directement avec les fichiers compilés sans les sources

Une fois que vous êtes sûrs de vos fichiers et que vous avez réalisés tous les tests nécessaires en contrôlant et en validant le bon déroulement de chacune des fonctions, vous n'êtes alors plus obligés de placer les sources dans votre projet. Cela vous évite de faire de mauvaises manipulations et vous permet également de cacher l'implémentation interne. Il suffit de placer dans votre projet, uniquement les fichiers compilés, puisque en définitive, l'éditeur de lien ne se sert que de ceux là.

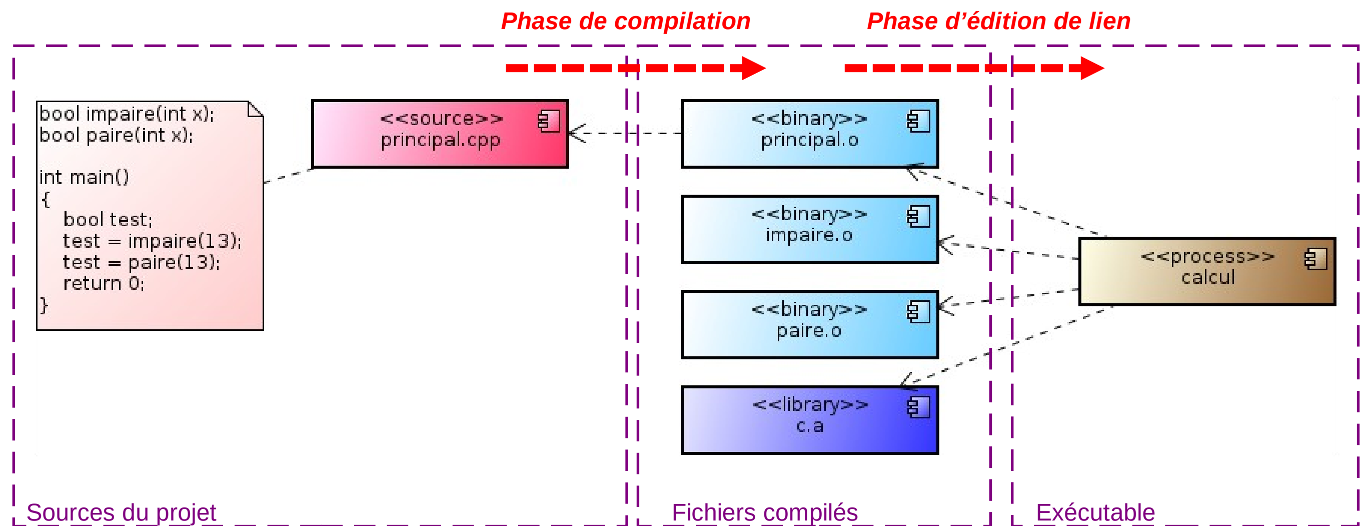


## Les bibliothèques

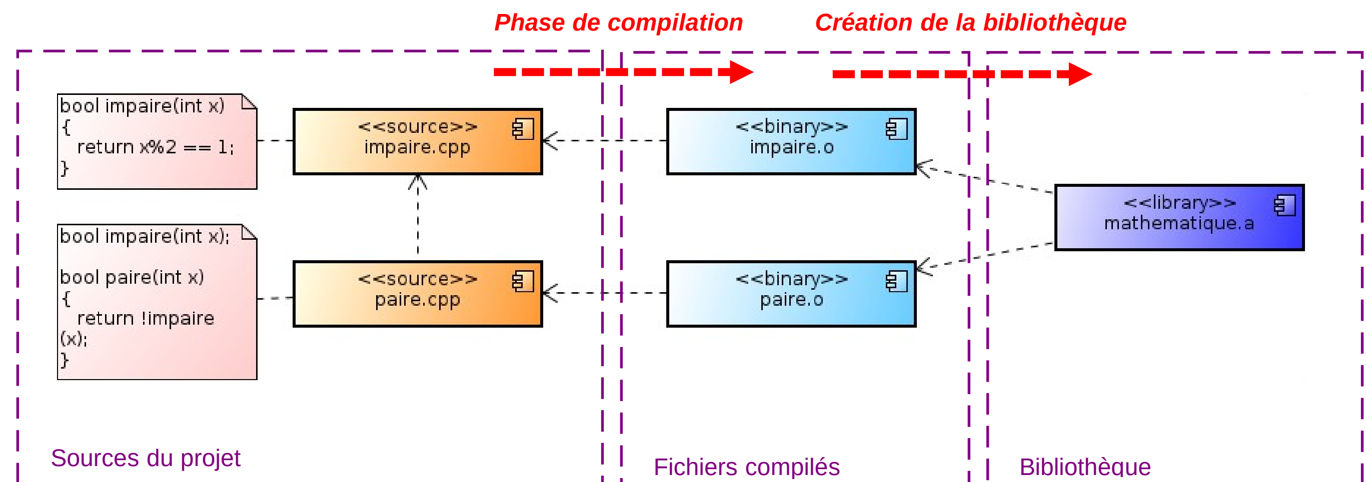
Il arrive assez souvent que les fonctions que nous construisons soient utiles pour d'autres projets. Il suffit de procéder de la même manière en plaçant les fichiers compilés dans le nouveau projet. N'oubliez pas de faire les déclarations nécessaires dans votre programme principal.

Par contre, la manipulation peut être fastidieuse si vous devez placer de nombreux fichiers compilés dans votre projet. Il est préférable, dans ce cas là, de fabriquer une bibliothèque qui rassemble les fichiers compilés dans un seul et unique fichier. Cette façon de procéder est intéressante, surtout si les fichiers compilés correspondent à une même rubrique. C'est justement notre cas. Les fichiers annexes ***« impaire.o »*** et ***« paire.o »*** représentent des fonctions mathématiques.

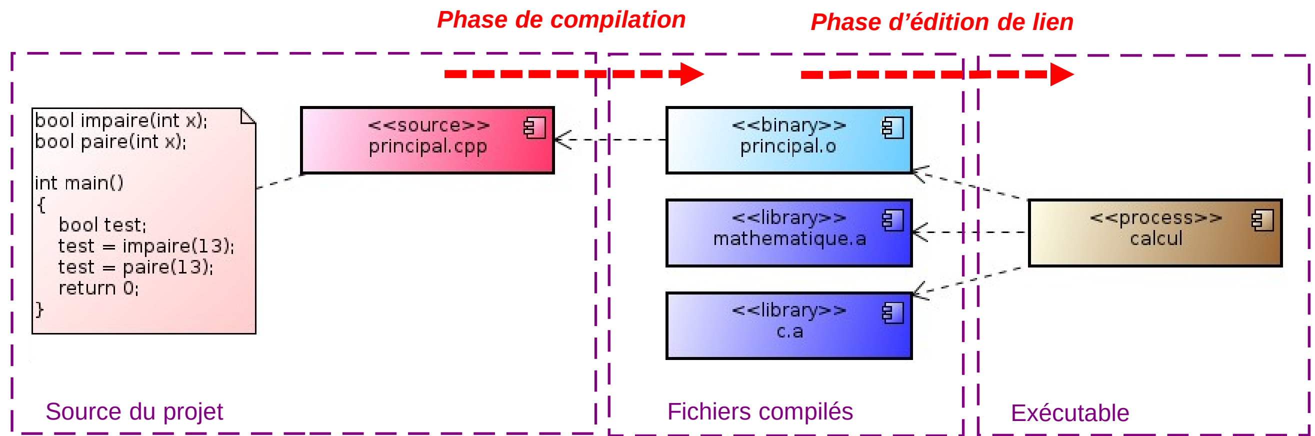
Les bibliothèques sont aussi souvent appelées des librairies. D'ailleurs, l'extension proposée est ***« *.a »***. A ce sujet, il faut savoir que systématiquement, les outils de développement intégrés, place au moins une librairie qui stocke toutes les définitions des fonctions standards du langage C++ comme les fonctions d'affichage, de saisie, etc. Cette librairie s'appelle ***« c.a »*** (suivant les outils, le nom peut être différent, mais il y a toujours la lettre c dedans). C'est notamment pour cette raison que la taille du fichier exécutable peut sembler conséquente.




Les environnements de développement intégrés permettent de créer de nouvelles bibliothèques. Dans le projet, au lieu de fabriquer un exécutable, nous demandons à la place de fabriquer une bibliothèque. Dans ce cas là, soit nous utilisons les fichiers déjà compilés, soit nous démarrons avec les sources qui seront donc compilés avant d'être intégrés dans la bibliothèque. Dans notre exemple, nous allons mettre en œuvre une bibliothèque ***« mathématique »***.




Une fois que la bibliothèque est créée, il est possible de l'utiliser dans n'importe quel projet. Nous allons d'ailleurs intégrer cette bibliothèque dans le projet de départ pour bien voir la différence.




## Le préprocesseur

Grâce à toutes ces techniques, nous avons beaucoup progressé dans l'élaboration d'un projet. La situation est devenue assez satisfaisante, mais pas tout a fait. Imaginons que nous ayons besoin d'une dizaine de fonctions issues d'une bibliothèque. Nous sommes obligés de déclarer systématiquement ces dix fonctions à chacun des fichiers que nous développons. Cette démarche devient fastidieuse. Il est préférable que ces déclarations soient faites automatiquement en liaison avec la bibliothèque utilisée.

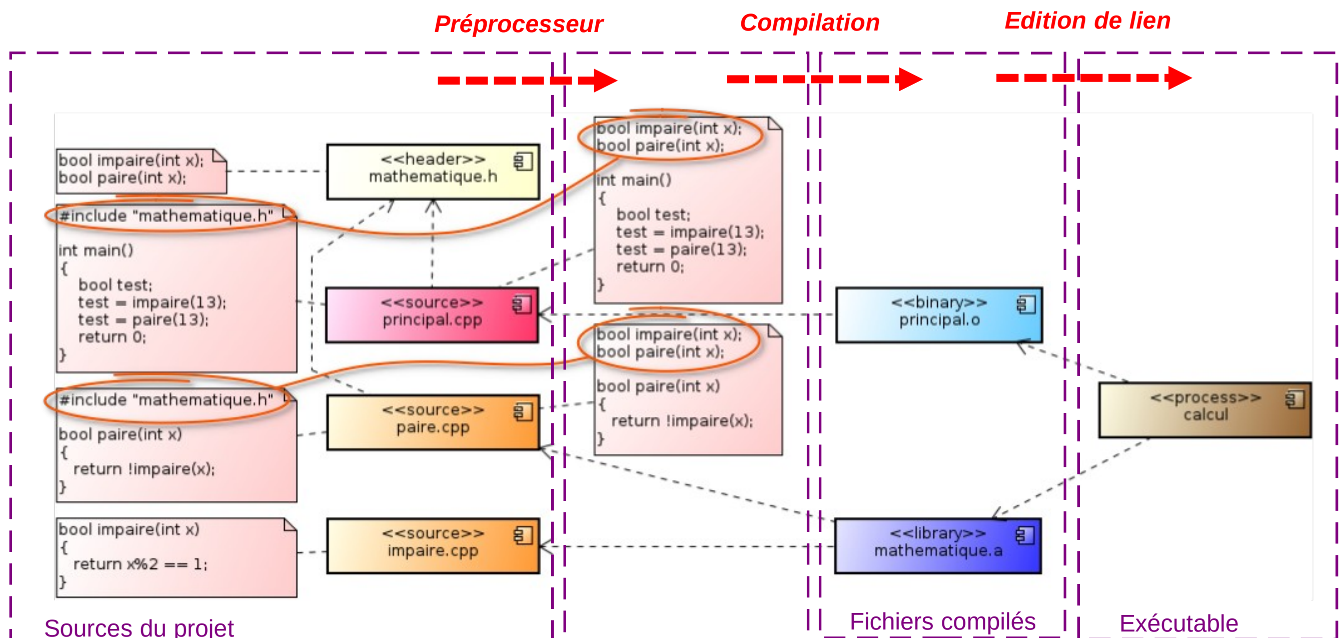
### Les fichiers en-têtes :

Il existe un fichier qui s'occupe de toutes les déclarations nécessaires à une bibliothèque. Il s'agit des fichiers en-têtes. Généralement, le nom du fichier inclus porte le même nom que la bibliothèque suivi de l'extension ***« *.h »*** (***header*** – *en-tête)*. Toutefois, si la librairie est conséquente, il est possible d'avoir plusieurs fichiers en-têtes qui sont établis par thème d'étude. C'est le cas avec le fichier en-tête ***« iostream.h »*** qui ne dispose que des déclarations relatives aux entrées-sorties.

### Directive d'inclusion :

Il faut maintenant que le contenu du fichier en-tête soit copié automatiquement dans le fichier source requérant ces déclarations. Il s'agit d'une copie de texte, et c'est le préprocesseur qui s'occupe de ce genre d'intervention. Dans ce cas, il faut indiquer au préprocesseur le traitement à réaliser en utilisant des directives appropriés. Les directives du préprocesseur sont spécifiées par un dièse ***« # »*** dans la toute première colonne d'une ligne de programme. Pour l'inclusion, il faut utiliser la directive ***« #include »*** suivi du nom du fichier à inclure. Il existe trois syntaxes :

1. **#include « iostream.h » :** le préprocesseur recherche le fichier à inclure dans le répertoire du projet ou le répertoire courant.
2. **#include <iostream.h> :** le préprocesseur recherche le fichier à inclure dans le répertoire prédéfini pour tous les fichiers en-têtes. Il s'agit du répertoire ***« include »*** qui est créé systématiquement par tous les outils de développement.
3. **#include <iostream> :** la recherche est la même que précédemment, mais le préprocesseur prend en compte les espaces de noms (ce sujet sera traité ultérieurement).



Pour notre projet, nous fabriquons un fichier en-tête ***« mathématique.h »*** relatif à la bibliothèque ***« mathématique.a »*** qui comporte les déclarations des deux fonctions. Il faut penser à changer le fichier source de la fonction ***paire()*** parce qu'elle-même fait référence à la fonction ***impaire()*** et doit donc disposer de sa déclaration.
Le fait d'utiliser des fichiers en-têtes nous évite de se souvenir de toutes les déclarations. Il suffit de placer la directive d'inclusion sans aucun tracas particulier. Il est vrai que nous aurions pu laisser la déclaration de la fonction ***impaire()*** dans le fichier source de la fonction ***paire()*** plutôt que de proposer l'inclusion. En effet, seule la déclaration de la fonction ***impaire()*** nous intéresse à ce moment là. A vous de choisir la meilleure opportunité.

**Les autres directives de compilation :**

Dans cet exemple simplifié au maximum, vous remarquez malgré tout qu'un même fichier en-tête peut être sollicité plusieurs fois. Du coup, le temps de compilation peut augmenter considérablement par ces ouvertures successives. Surtout pour relire systématiquement les mêmes choses alors que le préprocesseur, lui, est capable de mémoriser les déclarations qui ont déjà été faites. Il est souhaitable de prévenir ce genre de problème en proposant des compilations conditionnelles.

| ***Constantes symboliques*** | |
|---|---|
| **#define identificateur** | Permet de définir un paramètre de nom ***identificateur*** qui pourra être utilisé dans une clause **#if**. Tant que le préprocesseur n'est pas passé sur cette ligne, l'identificateur n'est pas encore connu. Par contre, après lecture de cette ligne, cet identificateur est définitivement validé (sauf avis contraire grâce à la directive **#undef**). |
| **#define PI 3.141592** | Sert à effectuer un changement de texte en remplaçant un symbole par un autre ou par une constante. Chaque fois que le symbole *'PI'* sera rencontré, le préprocesseur le remplacera par la constante ***'3.141592'.*** il est toutefois préférable d'utiliser ***const*** pour gérer les constantes. |
| ***Compilation conditionnelle*** | |
| Les directives conditionnelles permettent d'incorporer ou d'exclure de la compilation des portions de texte de programme selon que l'évaluation de la condition donne vrai ou faux comme résultat. | |
| **#ifdef identificateur** | Inclusion du texte qui suit cette ligne si l'identificateur est connu, c'est-à-dire s'il a déjà été défini. |
| **#ifndef identificateur** | Inclusion du texte qui suit cette ligne si l'identificateur n'est pas encore connu. |
| **#else** | Clause sinon associée à ***#ifdef*** ou à ***#ifndef*** |
| **#endif** | Fin du si associé à ***#ifdef*** ou à ***#ifndef*** |
| **#undef** | Met fin à l'existence d'un identificateur associé à un ***#define*** |

Pour être sûr que notre fichier en-tête soit lu qu'une seule fois, nous allons donc lui spécifier une compilation conditionnelle.

mathematique.h | Principal.cpp

```
#ifndef MatemathiqueH
#define MatemathiqueH

bool impaire(int x);
bool paire(int x);

#endif
```

**Le préprocesseur contrôle si l'identificateur 'MathematiqueH' est déjà connu. Il ne peut être connu qui si une définition lui étant associé a déjà été rencontrée. Ce qui n'est pas le cas la première fois. Du coup, tout se qui suit va être interprété par le préprocesseur jusqu'à ce qu'il rencontre *#endif*.**

**La première ligne que rencontre le préprocesseur après le test est justement la définition de l'identificateur. Ce qui fait que cette fois, l'identificateur est définitivement validé. Du coup, toute la partie entre *#ifndef* et *#endif* ne sera lu qu'une seule fois. Remarquez le nom proposé pour l'identificateur, il correspond au nom du fichier. C'est généralement la pratique utilisée.**

## Constitution générale d'un fichier en-tête :

***Attention***

Ces fichiers ne doivent absolument pas contenir de définition, seulement des déclarations, ou tout ce qui se rapporte à du traitement de texte comme, par exemple, les fonctions ***« en ligne »***.

Il est donc possible d'avoir :

- Des inclusions.
- Des énumérations (attention : sans proposer de nom de variable à la suite).
- Des structures (également, sans associer de nom de variables).
- Des déclarations de fonctions.
- Des définitions de fonctions en ligne puisque, dans ce cas, le préprocesseur effectuera du traitement de texte.
- Des définitions de fonctions ou de classes paramétrées (***génériques***). Ce sujet sera traité ultérieurement. Là aussi, c'est du traitement de texte.
- Des déclarations de variables (sans définition). Jusqu'à présent, nous avons toujours employé le terme ***déclaration de variable***, alors qu'il s'agit en fait d'une ***définition***. En effet, au moment d'une ***déclaration*** conventionnelle, un emplacement mémoire est réservé à la variable. Donc elle existe, ou si vous voulez, elle est définie. Une variable simplement déclarée comporte le préfixe ***extern***.
- Les espaces de nom.
- Les classes (qui seront traitées lors de la programmation orientée objet).

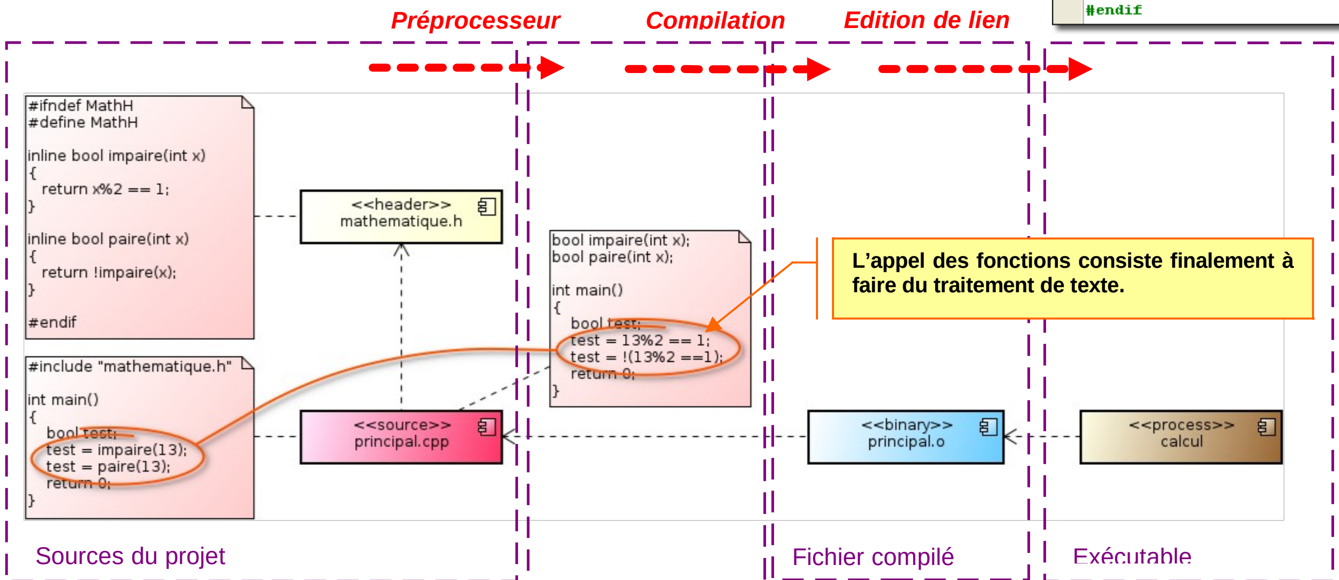
```
#ifndef UnFichierQuelconqueH
#define UnFichierQuelconqueH

#include <UnAutreFichier.h>

const double PI = 3.1415926;

enum Mois {Janvier, Fevrier, ..., Decembre};

struct Date {
  unsigned short jour;
  Mois mois;
  int annee;
};

bool impaire(int);

inline int minimum(int x, int y)
{
  return x<=y ? x : y;
}

template<class Type>
inline Type minimum(Type x, Type y)
{
  return x<=y ? x : y;
}

extern int i;

namespace EspaceDeNom {
  ...
}

class Complexe {
  double reel, imaginaire;
  double module();
  double argument();
};

#endif
```

Avec ce qui vient d'être dit, vous avez un aperçu de ce que nous ne devons absolument pas retrouver dans un fichier en-tête.

```
#ifndef UnFichierQuelconqueH
#define UnFichierQuelconqueH

int i;

struct Date {
  unsigned short jour;
  Mois mois;
  int annee;
} naissance;

int minimum(int x, int y)
{
  return x<=y ? x : y;
}

#endif
```

### Fonctions « inline » :

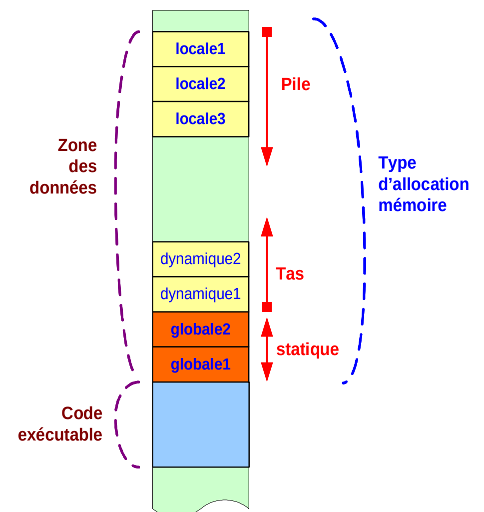
Pour notre projet, nous allons changer de stratégie. Comme les fonctions utilisées sont très réduites, il est souhaitable de les écrire en ligne. Elles vont donc se retrouver dans le fichier en-tête. Nous aurions pu le prévoir dès le commencement, mais le but poursuivi était de bien maîtriser les mécanismes des projets multi fichiers.




## Portée et durée de vie des variables

Il existe trois types de variables. Tout dépend de la manière de les déclarer et où elles sont déclarées. (Il s'agit ici, comme nous l'avons déjà évoqué, de déclaration au sens large, c'est-à-dire que ces variables, quelque soient leurs types, seront systématiquement définies). Par ailleurs, à chacune de ces variables correspondra une zone d'allocation particulière.

1. **Les variables locales (automatiques) :** Ceux sont toutes les variables qui sont déclarées à l'intérieur des fonctions ou en tant que paramètres. On dit que la portée de ces variables est la fonction. Du coup, la durée de vie est limitée à la durée de vie de la fonction. Attention, une variable locale qui n'est pas explicitement initialisée contient une valeur aléatoire (c'est ce qu'il se trouvait au préalable dans cette zone mémoire). Il peut exister des variables locales dont la durée de vie est encore plus courte. Il suffit que leurs portées soient limitées à un bloc d'instruction, comme par exemple, pour les boucles for où généralement nous déclarons une variable dans la zone d'initialisation qui gère l'index de l'itérative. Cette variable n'existe que pendant le traitement de la boucle. En dehors, elle n'est plus accessible puisqu'elle n'existe plus. La zone d'allocation de toutes les variables locales est la pile d'exécution.
2. **Les variables globales :** Cela doit être exceptionnel, mais nous pouvons avoir besoin d'une variable qui soit accessible depuis n'importe quelle fonction. Elle doit donc être déclarée en dehors de toute fonction. On dit alors que cette variable est globale. Dans ce cas là, la durée de vie correspond à la durée de vie du programme. Par ailleurs, la déclaration s'effectue dans la zone d'allocation statique. Il est bien entendu possible d'initialiser une variable globale avec une valeur particulière. Toutefois, sans initialisation explicite, une variable globale prend systématiquement la valeur 0.
3. **Les variables dynamiques :** cette fois ci, la durée de vie de ces variables est entièrement gérée par le programmeur. C'est justement l'intérêt de ce type de variable. Leurs existences commencent avec l'opérateur 'new' et se termine par l'opérateur 'delete'. Ces opérateurs peuvent d'ailleurs être situés dans des fonctions différentes. La zone d'allocation n'est ni la zone statique, ni la pile. C'est dans la zone mémoire qui se trouve entre les deux et qui s'appelle le tas (heap). Cette zone mémoire est une réserve dans laquelle nous pouvons y puiser durant l'exécution du programme (d'où son nom).

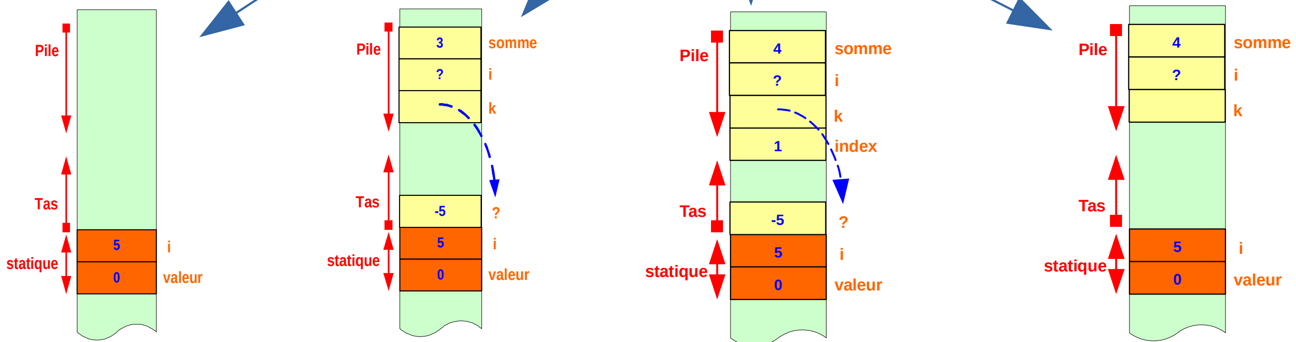



### Gestion de la pile et du tas :

La pile évolue des adresses hautes vers les adresses basses (les variables locales déposées sur la pile le sont en commençant par le haut) tandis que le tas évolue des adresses basses vers les adresses hautes. Cette technique présente l'avantage que, pour une même zone de mémoire allouée à l'ensemble « pile+tas », nous pourrons avoir, selon le moment de l'exécution du programme, soit une pile importante et un tas limité, soit une pile limitée et un tas important.

Une situation de dépassement de capacité de pile survient lorsque le sommet de la pile rejoint le sommet du tas.

```
//------------------------------------------------------------
int valeur;                    // variables globales
int i = 5;                     // définies dans la zone statique
//------------------------------------------------------------
int main()
{
  int somme = 3;               // variables locales à la fonction main
  int i;                       // définies sur la pile

  int *k = new int(-5);        // variable dynamique définie sur le tas

  for (int index=1; index<5; index++)
    somme += index;            // accès à la variable index locale à for
  valeur = 3;
  i = 2;                       // modification de i locale à main
  ::i = 12;                    // modification de la variable globale
  delete k;
  return 0;
}
//------------------------------------------------------------
```




**Résolution des portées de variable :**

Dans cet exemple, les variables utilisées sont déclarées dans des portées différentes. A ce sujet, la variable index est dans une portée extrêmement limitée puisqu'elle n'existe que durant la mise en œuvre de l'itérative for. Cette variable est une variable locale, elle est donc située sur la pile. Nous avons fréquemment des imbrications de portées locales (une portée à l'intérieur d'une autre portée).

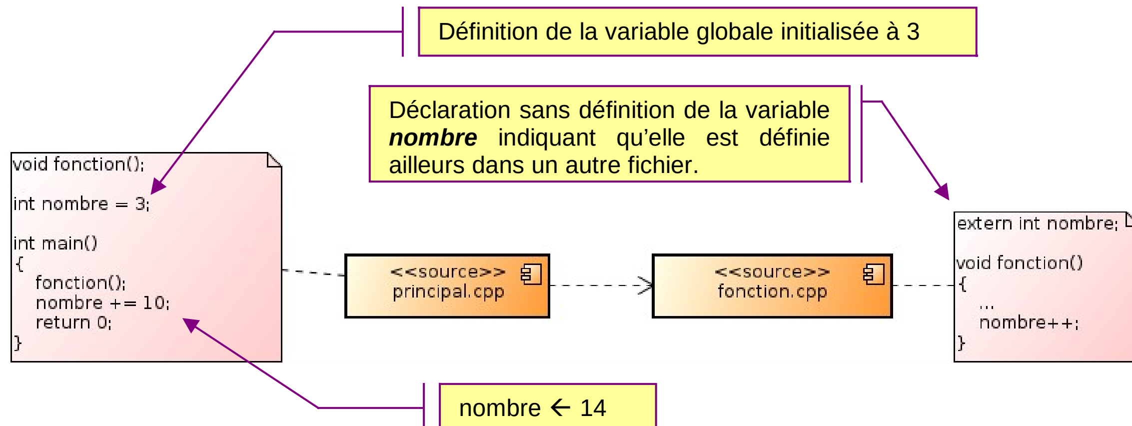
La résolution de nom dans une portée locale se déroule ainsi : la portée immédiate dans laquelle le nom est utilisé est recherchée. Si la déclaration est trouvée, le nom est déterminé ; sinon, la portée englobante est examinée. Ce processus se poursuit jusqu'à ce qu'une déclaration soit trouvée ou que la portée globale soit examinée. Dans ce cas et si aucune déclaration n'est trouvée pour le nom, l'utilisation du nom provoque une erreur de compilation.

Grâce à ce système de résolution de nom, chaque portée peut définir ses variables comme elle l'entend sans se soucier du nom. Il est alors possible de déclarer plusieurs fois le même nom de variable dans des portées différentes. A cause de l'ordre dans lequel les portées sont examinées lors de la résolution de nom, une déclaration dans une portée englobante se retrouve cachée par une déclaration du même nom déclarée dans une portée imbriquée.

Dans notre exemple, nous avons effectivement déclarés deux variables ***i***, dont la première est une variable globale et la seconde est une variable locale (Nous aurions même pu choisir une troisième variable ***i*** comme nom de compteur d'itérative au lieu d'index, puisque la portée est également différente). Du coup, au sein de la fonction, il n'est plus possible d'atteindre la variable globale puisqu'elle est cachée par la variable ***i*** locale. Si, malgré tout, nous désirons communiquer avec la variable ***i*** globale, il faut alors utiliser l'opérateur de portée **« :: »**. Il est évident que pour éviter ce genre de problème, il est souvent préférable de choisir des noms différents pour vos variables.

## Déclaration d'une variable sans la définir - extern

Dans un programme, nous avons quelque fois besoin d'une variable globale. Un problème se pose si notre programme est décomposé en plusieurs fichiers. En effet, nous savons que nous devons définir cette variable qu'une seule fois, sinon, nous avons une duplication de nom sur la même portée.



Sa définition va donc, nécessairement, se trouver dans un des fichiers. Malheureusement, lorsque nous nous trouvons dans un autre fichier, le nom de cette variable n'est plus visible. Pour que le compilateur ne soit pas perturbé, il faut alors préciser que la variable est déjà définie mais à l'extérieur de notre fichier. Dans ce cas de figure, il faut juste faire une déclaration (sans définition) en utilisant le préfixe ***« extern »***. Si plusieurs fichiers sont concernés par ce problème, il est alors préférable de réaliser la déclaration au sein d'un fichier en-tête ce qui permet de l'écrire qu'une seule fois.

#### Variable globale, variable locale

Ne serait-il pas plus simple de déclarer toutes les variables comme des variables globales ?

Déclarer en local les variables propres à une fonction vous fait d'abord économiser énormément de mémoire, puisque la même zone (la pile) est utilisée pour créer les variables locales de toutes les fonctions, en sachant, qu'à un instant donné, seules quelques variables existent. Le même emplacement mémoire est donc utilisé pour de nombreuses variables différentes. Dans le cas contraire, chaque variable devrait posséder sa propre case se qui représenterait une taille plus que conséquente.

Mais surtout, en utilisant des variables locales, vous rendez les programmes plus lisibles et plus fiables en associant intimement les variables à leurs fonctions.

Pourquoi « plus fiable » ?

Parce que vous évitez ainsi qu'une variable utilisée dans une fonction ne soit malencontreusement modifiée par une autre fonction. En effet, une variable globale est ***publique***. N'importe qui peut l'atteindre. Nous risquons donc de retrouver cette variable dans un état qui n'est pas prévu au moment où nous en avons besoin.

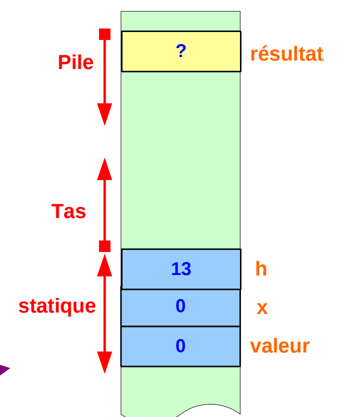
Cette notion de protection est fondamentale et, dans la mesure du possible, il est préférable de ne jamais utiliser de variable globale.

Malgré toutes ces remarques, une variable globale possède l'avantage de proposer la persistance. Sa durée de vie est la durée de vie du programme. Ainsi, il est possible de conserver une valeur indépendamment de la durée de vie de la fonction. Ce qui est dommage, c'est qu'elle soit ***publique***.

### Variables locales statiques - static

Heureusement, le langage C++ propose les variables statiques qui permettent de gérer la persistance tout en ayant un statut ***privé***. Une variable statique est une variable locale à une fonction, mais qui garde sa valeur d'une exécution à l'autre de la fonction. Elle est introduite par le préfixe ***« static »***. Du coup, cette variable utilise la zone d'allocation statique au lieu d'utiliser la pile. Bien que sa valeur persiste au cours des invocations de la fonction, la visibilité de son nom reste limitée à sa portée locale.

Comme la zone d'allocation est la zone statique, cette variable est systématiquement initialisée, soit par une valeur que vous devez préciser, soit automatiquement avec la valeur 0. Cette initialisation s'effectue uniquement lorsque l'exécution du programme passe sur la déclaration la première fois.




```
int valeur;
//------------------------
void fonction()
{
  int resultat;
  static int x;
  static int h = 13;
  ...
}
//------------------------
```

A la toute première évocation de la fonction, **x** ← 0, **h** ← 13. (**résultat** est quelconque puisqu'elle déclarée sur la pile). Lors des passages suivants, **x** et **h** conservent les valeurs qu'elles avaient au moment de quitter la fonction.