La programmation modulaire

Dans cette seconde partie, nous allons découvrir des concepts plus avancés du langage C. Je ne vous le cache pas, et vous vous en doutiez sûrement, la partie II est d'un cran de difficulté supérieur. Lorsque vous serez arrivés à la fin de cette partie, vous serez capables de vous débrouiller dans la plupart des programmes écrits en C. Dans la partie suivante nous verrons alors comment ouvrir une fenêtre, créer des jeux 2D, etc.

Jusqu'ici nous n'avons travaillé que dans un seul fichier appelé`main.c`. Pour le moment c'était acceptable car nos programmes étaient tout petits, mais ils vont bientôt être composés de dizaines, que dis-je de centaines de fonctions, et si vous les mettez toutes dans un même fichier celui-là va finir par devenir très long !
C'est pour cela que l'on a inventé ce qu'on appelle la programmation modulaire. Le principe est tout bête : plutôt que de placer tout le code de notre programme dans un seul fichier (`main.c`), nous le « séparons » en plusieurs petits fichiers.

Les prototypes

Jusqu'ici, je vous ai demandé de placer votre fonction avant la fonction`main`.
Pourquoi ?

Parce que l'ordre a une réelle importance ici : si vous mettez votre fonction avant le`main`dans votre code source, votre ordinateur l'aura lue et la connaîtra. Lorsque vous ferez un appel à la fonction dans le`main`, l'ordinateur connaîtra la fonction et saura où aller la chercher.
En revanche, si vous mettez votre fonction après le`main`, ça ne marchera pas car l'ordinateur ne connaîtra pas encore la fonction. Essayez, vous verrez !

Mais… c'est un peu mal fait, non ?

Tout à fait d'accord avec vous ! Mais rassurez-vous, les programmeurs s'en sont rendu compte avant vous et ont prévu le coup.

Grâce à ce que je vais vous apprendre maintenant, vous pourrez positionner vos fonctions dans n'importe quel ordre dans le code source. C'est mieux de ne pas avoir à s'en soucier, croyez-moi.

Le prototype pour annoncer une fonction

Nous allons « annoncer » nos fonctions à l'ordinateur en écrivant ce qu'on appelle des **prototypes**. Ne soyez pas intimidés par ce nom *high-tech*, ça cache en fait quelque chose de très simple.

Regardez la première ligne de notre fonction`aireRectangle`:

```
double aireRectangle(double largeur, double hauteur)
{
    return largeur * hauteur;
}
```

Copiez la première ligne (`double aireRectangle…`) tout en haut de votre fichier source (juste après les`#include`).
Rajoutez un point-virgule à la fin de cette nouvelle ligne.
Et voilà ! Maintenant, vous pouvez placer votre fonction`aireRectangle`après la fonction`main`si vous le voulez !

Vous devriez avoir le code suivant sous les yeux :

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

// La ligne suivante est le prototype de la fonction aireRectangle :
double aireRectangle(double largeur, double hauteur);

int main(int argc, char *argv[])
{
    printf("Rectangle de largeur 5 et hauteur 10. Aire = %f\n",
aireRectangle(5, 10));
    printf("Rectangle de largeur 2.5 et hauteur 3.5. Aire = %f\n",
aireRectangle(2.5, 3.5));
    printf("Rectangle de largeur 4.2 et hauteur 9.7. Aire = %f\n",
aireRectangle(4.2, 9.7));

    return 0;
}

// Notre fonction aireRectangle peut maintenant être mise n'importe où dans
le code source :
double aireRectangle(double largeur, double hauteur)
{
    return largeur * hauteur;
}
```

Ce qui a changé ici, c'est l'ajout du prototype en haut du code source.
Un prototype, c'est en fait une indication pour l'ordinateur. Cela lui indique qu'il existe une fonction appelée`aireRectangle`qui prend tels paramètres en entrée et renvoie une sortie du type que vous indiquez. Cela permet à l'ordinateur de s'organiser.

Grâce à cette ligne, vous pouvez maintenant placer vos fonctions dans n'importe quel ordre sans vous prendre la tête !

Écrivez toujours les prototypes de vos fonctions. Vos programmes ne vont pas tarder à se complexifier et à utiliser de nombreuses fonctions : mieux vaut prendre dès maintenant la bonne habitude d'écrire le prototype de chacune d'elles.

Comme vous le voyez, la fonction`main`n'a pas de prototype. En fait, c'est la seule qui n'en nécessite pas, parce que l'ordinateur la connaît (c'est toujours la même pour tous les programmes, alors il peut bien la connaître, à force !).

Pour être tout à fait exact, il faut savoir que dans la ligne du prototype il est facultatif d'écrire les noms de variables en entrée. L'ordinateur a juste besoin de connaître les types des variables.

On aurait donc pu simplement écrire :

```
double aireRectangle(double, double);
```

Toutefois, l'autre méthode que je vous ai montrée tout à l'heure fonctionne aussi bien. L'avantage avec ma méthode, c'est que vous avez juste besoin de copier-coller la première ligne de la fonction et de rajouter un point-virgule. Ça va plus vite.

N'oubliez JAMAIS de mettre un point-virgule à la fin d'un prototype. C'est ce qui permet à l'ordinateur de différencier un prototype du véritable début d'une fonction.
Si vous ne le faites pas, vous risquez d'avoir des erreurs incompréhensibles lors de la compilation.

Les headers

Jusqu'ici, nous n'avions qu'un seul fichier source dans notre projet. Ce fichier source, je vous avais demandé de l'appeler`main.c`.

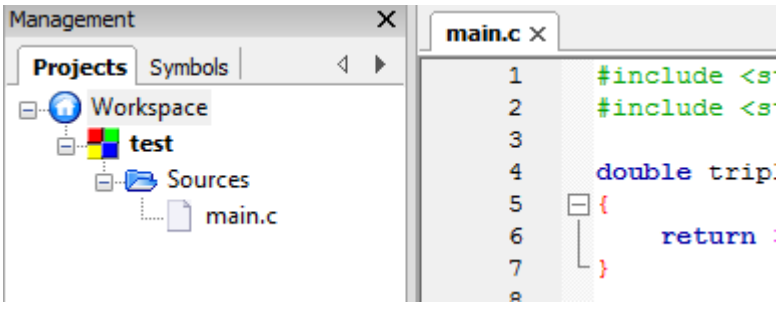
Plusieurs fichiers par projet

Dans la pratique, vos programmes ne seront pas tous écrits dans ce même fichier`main.c`. Bien sûr, il est *possible* de le faire, mais ce n'est jamais très pratique de se balader dans un fichier de 10 000 lignes (enfin, personnellement, je trouve !). C'est pour cela qu'en général on crée plusieurs fichiers par projet.

Qu'est-ce qu'un projet, déjà ?

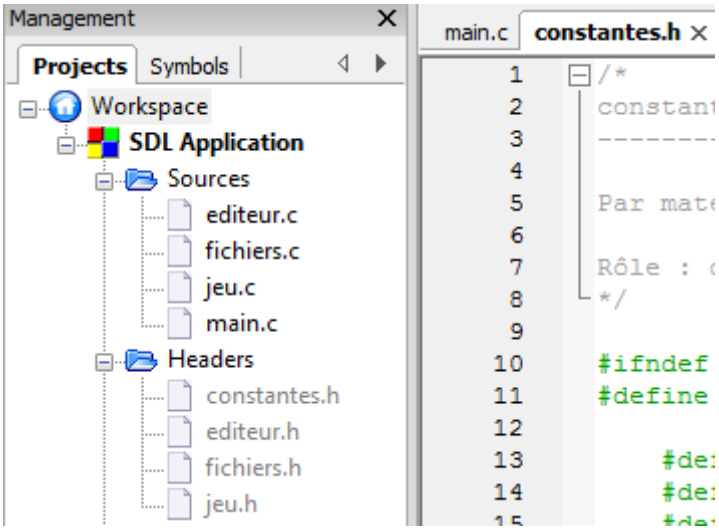
Non, vous n'avez pas déjà oublié ? Bon : je vous le réexplique, parce qu'il est important qu'on soit bien d'accord sur ce terme.

Un projet, c'est l'ensemble des fichiers source de votre programme. Pour le moment, nos projets n'étaient composés que d'un fichier source. Regardez dans votre IDE, généralement c'est sur la gauche (fig. suivante).



Comme vous pouvez le voir à gauche sur cette capture d'écran, ce projet n'est composé que d'un fichier`main.c`.

Laissez-moi maintenant vous montrer un vrai projet que vous réaliserez un peu plus loin dans le cours : un jeu de Sokoban (fig. suivante).



Comme vous le voyez, il y a plusieurs fichiers. Un vrai projet ressemblera à ça : vous verrez plusieurs fichiers dans la colonne de gauche. Vous reconnaissez dans la liste le fichier`main.c`: c'est celui qui contient la fonction`main`. En général dans mes programmes, je ne mets que le`main`dans`main.c`. Pour information, ce n'est pas du tout une obligation, chacun s'organise comme il veut. Pour bien me suivre, je vous conseille néanmoins de faire comme moi.

Mais pourquoi avoir créé plusieurs fichiers ? Et comment je sais combien de fichiers je dois créer pour mon projet ?

Ça, c'est vous qui choisissez. En général, on regroupe dans un même fichier des fonctions ayant le même thème. Ainsi, dans le fichier`editeur.c`j'ai regroupé toutes les fonctions concernant l'éditeur de niveau ; dans le fichier`jeu.c`, j'ai regroupé toutes les fonctions concernant le jeu lui-même, etc.

Fichiers`.h`et`.c`

Comme vous le voyez, il y a deux types de fichiers différents sur la fig. suivante.

- **Les`.h`**, appelés fichiers *headers*. Ces fichiers contiennent les prototypes des fonctions.
- **Les`.c`** : les fichiers source. Ces fichiers contiennent les fonctions elles-mêmes.

En général, on met donc rarement les prototypes dans les fichiers`.c`comme on l'a fait tout à l'heure dans le`main.c`(sauf si votre programme est tout petit).

Pour chaque fichier`.c`, il y a son équivalent`.h`qui contient les prototypes des fonctions. Jetez un œil plus attentif à la fig. suivante :

- il y a`editeur.c`(le code des fonctions) et`editeur.h`(les prototypes des fonctions) ;
- il y a`jeu.c`et`jeu.h`;
- etc.

Mais comment faire pour que l'ordinateur sache que les prototypes sont dans un autre fichier que le`.c`?

Il faut inclure le fichier `.h` grâce à une directive de préprocesseur.
Attention, préparez-vous à comprendre beaucoup de choses d'un coup !

Comment inclure un fichier header ?… Vous savez le faire, vous l'avez déjà fait !

Regardez par exemple le début de mon fichier `jeu.c` :

```
#include <stdlib.h>
#include <stdio.h>
#include "jeu.h"

void jouer(SDL_Surface* ecran)
{
// ...
```

L'inclusion se fait grâce à la directive de préprocesseur `#include` que vous connaissez bien maintenant. Regardez les premières lignes du code source ci-dessus :

```
#include <stdlib.h>
#include <stdio.h>
#include "jeu.h" // On inclut jeu.h
```

On inclut trois fichiers `.h` : `stdio`, `stdlib` et `jeu`.
Notez une différence : les fichiers que vous avez créés et placés dans le répertoire de votre projet doivent être inclus avec des guillemets (`"jeu.h"`) tandis que les fichiers correspondant aux bibliothèques (qui sont généralement installés, eux, dans le répertoire de votre IDE) sont inclus entre chevrons (`<stdio.h>`).

Vous utiliserez donc :

- **les chevrons `< >`** pour inclure un fichier se trouvant dans le répertoire « include » de votre IDE ;
- **les guillemets `" "`** pour inclure un fichier se trouvant dans le répertoire de votre projet (à côté des `.c`, généralement).

La commande `#include` demande d'insérer le contenu du fichier dans le `.c`. C'est donc une commande qui dit « Insère ici le fichier `jeu.h` » par exemple.

Et dans le fichier `jeu.h`, que trouve-t-on ?
On trouve simplement les prototypes des fonctions du fichier `jeu.c` !

```
/*
jeu.h
-----

Par mateo21, pour Le Site du Zéro (www.siteduzero.com)

Rôle : prototypes des fonctions du jeu.
*/

    void jouer(SDL_Surface* ecran);
    void deplacerJoueur(int carte[][NB_BLOCS_HAUTEUR], SDL_Rect *pos, int
direction);
    void deplacerCaisse(int *premiereCase, int *secondeCase);
```

Voilà comment fonctionne un vrai projet !

Quel est l'intérêt de mettre les prototypes dans des fichiers `.h`?

La raison est en fait assez simple. Quand dans votre code vous faites appel à une fonction, votre ordinateur doit déjà la connaître, savoir combien de paramètres elle prend, etc. C'est à ça que sert un prototype : c'est le mode d'emploi de la fonction pour l'ordinateur.

Tout est une question d'ordre : si vous placez vos prototypes dans des `.h`(headers) inclus en haut des fichiers `.c`, votre ordinateur connaîtra le mode d'emploi de toutes vos fonctions *dès* le début de la lecture du fichier.

En faisant cela, vous n'aurez ainsi pas à vous soucier de l'ordre dans lequel les fonctions se trouvent dans vos fichiers `.c`. Si maintenant vous faites un petit programme contenant deux ou trois fonctions, vous vous rendrez peut-être compte que les prototypes semblent facultatifs (ça marche sans). Mais ça ne durera pas longtemps ! Dès que vous aurez un peu plus de fonctions, si vous ne mettez pas vos prototypes de fonctions dans des `.h`, la compilation échouera sans aucun doute.

Lorsque vous appellerez une fonction située dans`fonctions.c`depuis le fichier`main.c`, vous aurez besoin d'inclure les prototypes de`fonctions.c`dans`main.c`. Il faudra donc mettre un`#include "fonctions.h"`en haut de`main.c`.
Souvenez-vous de cette règle : à chaque fois que vous faites appel à une fonction X dans un fichier, il faut que vous ayez inclus les prototypes de cette fonction dans votre fichier. Cela permet au compilateur de vérifier si vous l'avez correctement appelée.

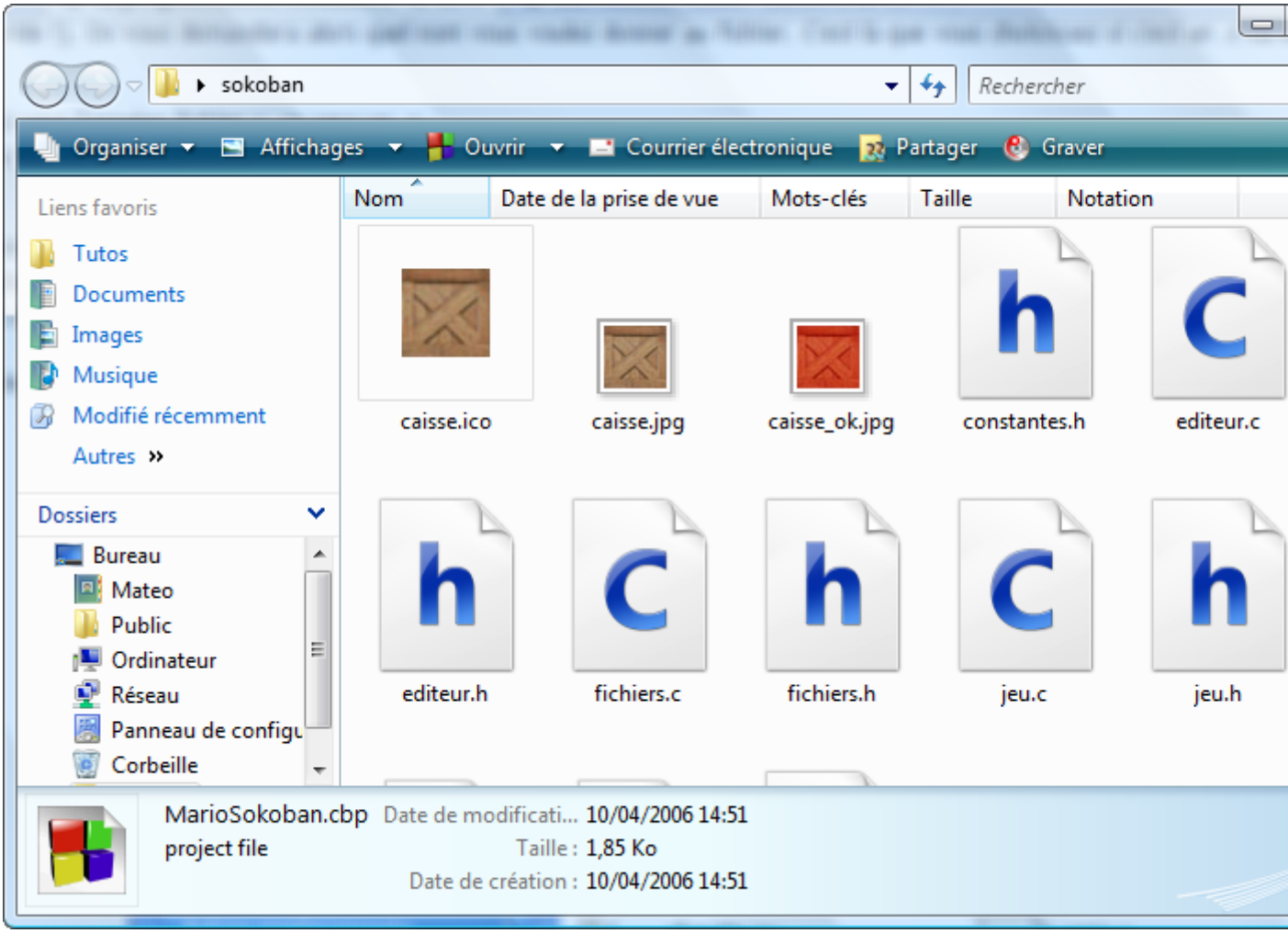
Comment puis-je ajouter des fichiers `.c`et `.h`à mon projet ?

Ça dépend de l'IDE que vous utilisez, mais globalement la procédure est la même :`Fichier / Nouveau / Fichier source`.
Cela crée un nouveau fichier vide. Ce fichier n'est pas encore de type `.c`ou `.h`, il faut que vous l'enregistriez pour le dire. Enregistrez donc ce nouveau fichier (même s'il est encore vide !). On vous demandera alors quel nom vous voulez donner au fichier. C'est là que vous choisissez si c'est un `.c`ou un `.h`:

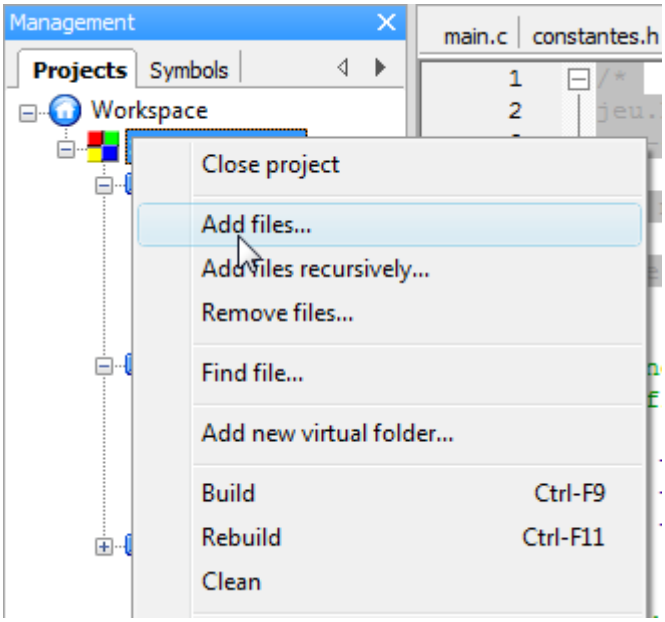
- si vous l'appelez`fichier.c`, ce sera un `.c`;
- si vous l'appelez`fichier.h`, ce sera un `.h`.

C'est aussi simple que cela. Enregistrez votre fichier dans le répertoire dans lequel se trouvent les autres fichiers de votre projet (le même dossier que`main.c`). Généralement, vous enregistrerez tous vos fichiers dans le même répertoire, les `.c`comme les `.h`.

Le dossier du projet ressemble au final à la fig. suivante. Vous y voyez des `.c`et des `.h`ensemble.



Votre fichier est maintenant enregistré, mais il n'est pas encore vraiment ajouté au projet ! Pour l'ajouter au projet, faites un clic droit dans la partie à gauche de l'écran (où il y a la liste des fichiers du projet) et choisissez`Add files`(fig. suivante).



Une fenêtre s'ouvre et vous demande quels fichiers ajouter au projet. Sélectionnez le fichier que vous venez de créer et c'est fait. Le fichier fait maintenant partie du projet et apparaît dans la liste à gauche !

Les `include` des bibliothèques standard

Une question devrait vous trotter dans la tête…
Si on inclut les fichiers `stdio.h` et `stdlib.h`, c'est donc qu'ils existent quelque part et qu'on peut aller les chercher, non ?

Oui, bien sûr !
Ils sont normalement installés là où se trouve votre IDE. Dans mon cas sous Code::Blocks, je les trouve là :

```
C:\Program Files\CodeBlocks\MinGW\include
```

Il faut généralement chercher un dossier `include`.
Là-dedans, vous allez trouver de très nombreux fichiers. Ce sont des headers (`.h`) des bibliothèques standard, c'est-à-dire des bibliothèques disponibles partout (que ce soit sous Windows, Mac, Linux…). Vous y retrouverez donc `stdio.h` et `stdlib.h`, entre autres.

Vous pouvez les ouvrir si vous voulez, mais ça risque de piquer un peu les yeux. En effet, c'est un peu compliqué (il y a pas mal de choses qu'on n'a pas encore vues, notamment certaines directives de préprocesseur). Si vous cherchez bien, vous verrez que ce fichier est rempli de prototypes de fonctions standard, comme `printf` par exemple.

Ok, je sais maintenant où se trouvent les prototypes des fonctions standard. Mais comment pourrais-je voir le code source de ces fonctions ? Où sont les `.c` ?

Vous ne les avez pas ! En fait, les fichiers `.c` sont déjà compilés (en code binaire, c'est-à-dire en code machine). Il est donc totalement impossible de les lire.

Vous pouvez retrouver les fichiers compilés dans un répertoire appelé `lib` (c'est l'abréviation de `library` qui signifie « bibliothèque » en français). Chez moi, on peut les trouver dans le répertoire :

```
C:\Program Files\CodeBlocks\MinGW\lib
```

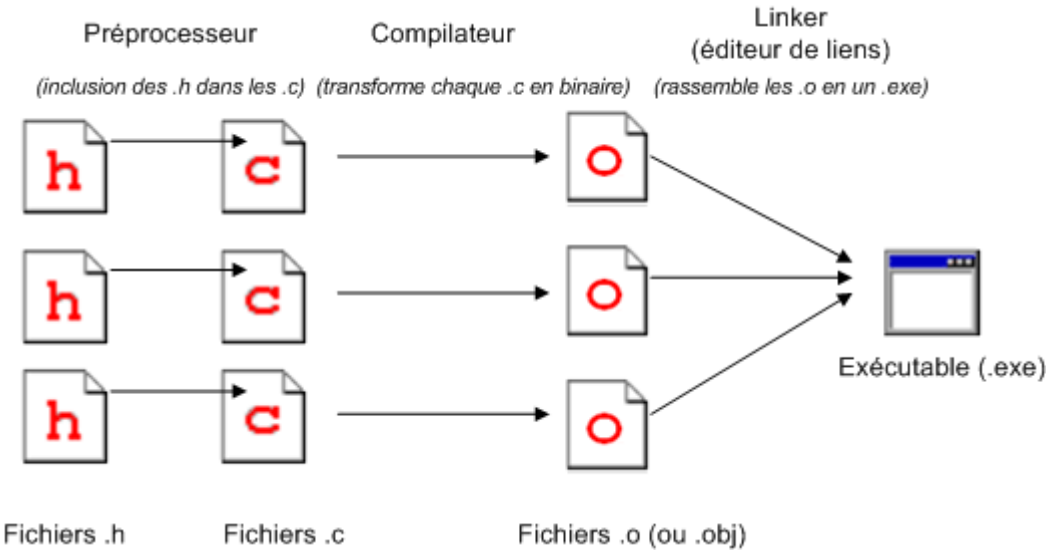
Les fichiers compilés des bibliothèques ont l'extension `.a` sous Code::Blocks (qui utilise le compilateur appelé `mingw`) et ont l'extension `.lib` sous Visual C++ (qui utilise le compilateur `Visual`). N'essayez pas de les lire : ce n'est absolument pas comestible pour un humain.

En résumé, dans vos fichiers `.c`, vous incluez les `.h` des bibliothèques standard pour pouvoir utiliser des fonctions standard comme `printf`. Votre ordinateur a ainsi les prototypes sous les yeux et peut vérifier si vous appelez les fonctions correctement, par exemple que vous n'oubliez pas de paramètres.

La compilation séparée

Maintenant que vous savez qu'un projet est composé de plusieurs fichiers source, nous pouvons rentrer plus en détail dans le fonctionnement de la compilation. Jusqu'ici, nous avions vu un schéma très simplifié.

La fig. suivante est un schéma bien plus précis de la compilation. C'est le genre de schémas qu'il est fortement conseillé de comprendre et de connaître par cœur !



Ça, c'est un vrai schéma de ce qu'il se passe à la compilation. Détaillons-le.

1.
    1. **Préprocesseur** : le préprocesseur est un programme qui démarre avant la compilation. Son rôle est d'exécuter les instructions spéciales qu'on lui a données dans des directives de préprocesseur, ces fameuses lignes qui commencent par un`#`.

Pour l'instant, la seule directive de préprocesseur que l'on connaît est`#include`, qui permet d'inclure un fichier dans un autre. Le préprocesseur sait faire d'autres choses, mais ça, nous le verrons plus tard. Le`#include`est quand même ce qu'il y a de plus important à connaître.Le préprocesseur « remplace » donc les lignes`#include`par le fichier indiqué. Il met à l'intérieur de chaque fichier`.c`le contenu des fichiers`.h`qu'on a demandé d'inclure.À ce moment-là de la compilation, votre fichier`.c`est complet et contient tous les prototypes des fonctions que vous utilisez (votre fichier`.c`est donc un peu plus gros que la normale).

1.
    1. **Compilation** : cette étape très importante consiste à transformer vos fichiers source en code binaire compréhensible par l'ordinateur. Le compilateur compile chaque fichier`.c`un à un. Il compile tous les fichiers source de votre projet, d'où l'importance d'avoir bien ajouté tous vos fichiers au projet (ils doivent tous apparaître dans la fameuse liste à gauche).

Le compilateur génère un fichier`.o`(ou`.obj`, ça dépend du compilateur) par fichier`.c`compilé. Ce sont des fichiers binaires temporaires. Généralement, ces fichiers sont supprimés à la fin de la compilation, mais selon les options de votre IDE, vous pouvez choisir de les conserver. Bien qu'inutiles puisque temporaires, on peut trouver un intérêt à conserver les`.o`. En effet, si

parmi les 10 fichiers `.c` de votre projet seul l'un d'eux a changé depuis la dernière compilation, le compilateur n'aura qu'à recompiler seulement ce fichier `.c`. Pour les autres, il possède déjà les `.o` compilés.

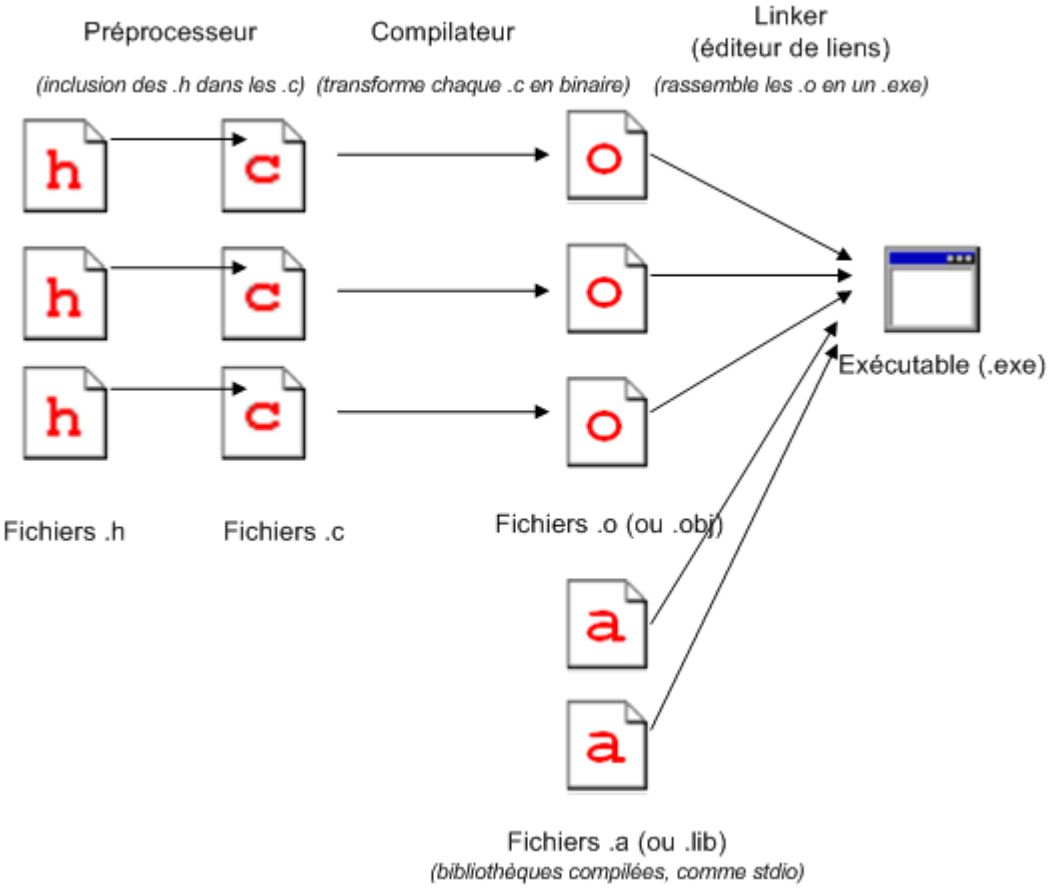
1. **Édition de liens** : le *linker* (ou « éditeur de liens » en français) est un programme dont le rôle est d'assembler les fichiers binaires `.o`. Il les assemble en un seul gros fichier : l'exécutable final ! Cet exécutable a l'extension `.exe` sous Windows. Si vous êtes sous un autre OS, il devrait prendre l'extension adéquate.

Maintenant, vous savez comment ça se passe à l'intérieur. Je le dis et je le répète, ce schéma de la fig. suivante est très important. Il fait la différence entre un programmeur du dimanche qui copie sans comprendre des codes source et un autre qui sait et comprend ce qu'il fait.

La plupart des erreurs surviennent à la compilation, mais il m'est aussi arrivé d'avoir des erreurs de linker. Cela signifie que le linker n'est pas arrivé à assembler tous les `.o` (il en manquait peut-être).

Notre schéma est par contre encore un peu incomplet. En effet, les bibliothèques n'y apparaissent pas ! Comment cela se passe-t-il quand on utilise des bibliothèques ?

En fait, le début du schéma reste le même, c'est seulement le linker qui va avoir un peu plus de travail. Il va assembler vos `.o` (temporaires) avec les bibliothèques compilées dont vous avez besoin (`.a` ou `.lib` selon le compilateur). La fig. suivante est donc une version améliorée de la fig. précédente.



Nous y sommes, le schéma est cette fois complet. Vos fichiers de bibliothèques `.a` (ou `.lib`) sont rassemblés dans l'exécutable avec vos `.o`.

C'est comme cela qu'on peut obtenir au final un programme 100 % complet, qui contient toutes les instructions nécessaires à l'ordinateur, même celles qui lui expliquent comment afficher du texte !
Par exemple, la fonction `printf` se trouve dans un `.a`, et sera donc rassemblée avec votre code source dans l'exécutable.

Dans quelque temps, nous apprendrons à utiliser des bibliothèques graphiques. Celles-ci seront là aussi dans des `.a` et contiendront des instructions pour indiquer à l'ordinateur comment ouvrir une fenêtre à l'écran, par exemple. Mais patience, car tout vient à point à qui sait attendre, c'est bien connu.

La portée des fonctions et variables

Pour clore ce chapitre, il nous faut impérativement découvrir la notion de **portée** des fonctions et des variables. Nous allons voir quand les variables et les fonctions sont accessibles, c'est-à-dire quand on peut faire appel à elles.

Les variables propres aux fonctions

Lorsque vous déclarez une variable dans une fonction, celle-ci est supprimée de la mémoire à la fin de la fonction :

```
int triple(int nombre)
{
    int resultat = 0; // La variable resultat est créée en mémoire

    resultat = 3 * nombre;
    return resultat;
} // La fonction est terminée, la variable resultat est supprimée de la
mémoire
```

Une variable déclarée dans une fonction n'existe donc que pendant que la fonction est exécutée.
Qu'est-ce que ça veut dire, concrètement ? Que vous ne pouvez pas y accéder depuis une autre fonction !

```
int triple(int nombre);

int main(int argc, char *argv[])
{
    printf("Le triple de 15 est %d\n", triple(15));

    printf("Le triple de 15 est %d", resultat); // Erreur

    return 0;
}

int triple(int nombre)
{
    int resultat = 0;
```

```
    resultat = 3 * nombre;
    return resultat;
}
```

Dans le`main`, j'essaie ici d'accéder à la variable`resultat`. Or, comme cette variable`resultat`a été créée dans la fonction`triple`, elle n'est pas accessible dans la fonction`main`!

Retenez bien : une variable déclarée dans une fonction n'est accessible qu'à l'intérieur de cette fonction. On dit que c'est une variable locale.

Les variables globales : à éviter

Variable globale accessible dans tous les fichiers

Il est possible de déclarer des variables qui seront accessibles dans toutes les fonctions de tous les fichiers du projet. Je vais vous montrer comment faire pour que vous sachiez que ça existe, mais généralement il faut éviter de le faire. Ça aura l'air de simplifier votre code au début, mais ensuite vous risquez de vous retrouver avec de nombreuses variables accessibles partout, ce qui risquera de vous créer des soucis.

Pour déclarer une variable « globale » accessible partout, vous devez faire la déclaration de la variable en dehors des fonctions. Vous ferez généralement la déclaration tout en haut du fichier, après les`#include`.

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

int resultat = 0; // Déclaration de variable globale

void triple(int nombre); // Prototype de fonction

int main(int argc, char *argv[])
{
    triple(15); // On appelle la fonction triple, qui modifie la variable
globale resultat
    printf("Le triple de 15 est %d\n", resultat); // On a accès à resultat

    return 0;
}

void triple(int nombre)
{
    resultat = 3 * nombre;
}
```

Sur cet exemple, ma fonction`triple`ne renvoie plus rien (`void`). Elle se contente de modifier la variable globale`resultat`que la fonction`main`peut récupérer.

Ma variable`resultat`sera accessible dans tous les fichiers du projet, on pourra donc faire appel à elle dans TOUTES les fonctions du programme.

Ce type de choses est généralement à bannir dans un programme en C. Utilisez plutôt le retour de la fonction (`return`) pour renvoyer un résultat.

Variable globale accessible uniquement dans un fichier

La variable globale que nous venons de voir était accessible dans tous les fichiers du projet. Il est possible de la rendre accessible uniquement dans le fichier dans lequel elle se trouve. Ça reste une variable globale quand même, mais disons qu'elle n'est globale qu'aux fonctions de ce fichier, et non à toutes les fonctions du programme.

Pour créer une variable globale accessible uniquement dans un fichier, rajoutez simplement le mot-clé`static`devant :

```
static int resultat = 0;
```

Variable statique à une fonction

Attention : c'est un peu plus délicat, ici. Si vous rajoutez le mot-clé`static`devant la déclaration d'une variable à l'intérieur d'une fonction, ça n'a pas le même sens que pour les variables globales.
En fait, la variable`static`n'est plus supprimée à la fin de la fonction. La prochaine fois qu'on appellera la fonction, la variable aura conservé sa valeur.

Par exemple :

```
int triple(int nombre)
{
    static int resultat = 0; // La variable resultat est créée la première
fois que la fonction est appelée

    resultat = 3 * nombre;
    return resultat;
} // La variable resultat n'est PAS supprimée lorsque la fonction est
terminée.
```

Qu'est-ce que ça signifie, concrètement ?
Qu'on pourra rappeler la fonction plus tard et la variable`resultat`contiendra toujours la valeur de la dernière fois.

Voici un petit exemple pour bien comprendre :

```
int incremente();

int main(int argc, char *argv[])
{
    printf("%d\n", incremente());
    printf("%d\n", incremente());
    printf("%d\n", incremente());
    printf("%d\n", incremente());

    return 0;
}

int incremente()
{
    static int nombre = 0;

    nombre++;
```

```
    return nombre;
}
1
2
3
4
```

Ici, la première fois qu'on appelle la fonction`incremente`, la variable`nombre`est créée. Elle est incrémentée à 1, et une fois la fonction terminée la variable n'est pas supprimée.

Lorsque la fonction est appelée une seconde fois, la ligne de la déclaration de variable est tout simplement « sautée ». On ne recrée pas la variable, on réutilise la variable qu'on avait déjà créée.
Comme la variable valait 1, elle vaudra maintenant 2, puis 3, puis 4, etc.

Les fonctions locales à un fichier

Pour en finir avec les portées, nous allons nous intéresser à la portée des fonctions. Normalement, quand vous créez une fonction, celle-ci est globale à tout le programme. Elle est accessible depuis n'importe quel autre fichier `.c`.

Il se peut que vous ayez besoin de créer des fonctions qui ne seront accessibles que dans le fichier dans lequel se trouve la fonction.

Pour faire cela, rajoutez le mot-clé`static`(encore lui) devant la fonction :

```
static int triple(int nombre)
{
    // Instructions
}
```

Pensez à mettre à jour le prototype aussi :

```
static int triple(int nombre);
```

Maintenant, votre fonction`static triple`ne peut être appelée que depuis une autre fonction du même fichier. Si vous essayez d'appeler la fonction`triple`depuis une fonction d'un autre fichier, ça ne marchera pas car`triple`n'y sera pas accessible.

Résumons tous les types de portée qui peuvent exister pour les variables :

- Une variable déclarée dans une fonction est supprimée à la fin de la fonction, elle n'est accessible que dans cette fonction.
- Une variable déclarée dans une fonction avec le mot-clé`static`devant n'est pas supprimée à la fin de la fonction, elle conserve sa valeur au fur et à mesure de l'exécution du programme.
- Une variable déclarée en dehors des fonctions est une variable globale, accessible depuis toutes les fonctions de tous les fichiers source du projet.
- Une variable globale avec le mot-clé`static`devant est globale uniquement dans le fichier dans lequel elle se trouve, elle n'est pas accessible depuis les fonctions des autres fichiers.

De même, voici les types de portée qui peuvent exister pour les fonctions :

- Une fonction est par défaut accessible depuis tous les fichiers du projet, on peut donc l'appeler depuis n'importe quel autre fichier.
- Si on veut qu'une fonction ne soit accessible que dans le fichier dans lequel elle se trouve, il faut rajouter le mot-clé`static`devant.

En résumé

- Un programme contient de nombreux fichiers `.c`. En règle générale, chaque fichier `.c`a un petit frère du même nom ayant l'extension `.h`(qui signifie **header**). Le `.c`contient les fonctions tandis que le `.h`contient les **prototypes**, c'est-à-dire la signature de ces fonctions.
- Le contenu des fichiers `.h`est inclus en haut des `.c`par un programme appelé **préprocesseur**.
- Les `.c`sont transformés en fichiers `.o`binaires par le **compilateur**.
- Les `.o`sont assemblés en un exécutable (`.exe`) par le **linker**, aussi appelé **éditeur de liens**.
- Une variable déclarée dans une fonction n'est pas accessible dans une autre fonction. On parle de **portée des variables.**