typedef

Definition (typedef)

Une déclaration qualifiée par **typedef** permet d'obtenir un alias de type, c'est-à-dire un identificateur associé au type correspondant. Syntaxe :

typedef *type identificateur*;

Cette construction est en général utilisée pour rendre lisible la définition d'un type ou pour abstraire un type particulier.

Exemple

```
#include <stdio.h>

typedef int element_t;
typedef element_t array_t[50];

int main(void) {
  array_t tableau;
}
```

Question

Que préférez-vous :

```
int (*(*t[32])(double (*)(int *,char)))[50];
```

ou

```
typedef int (*array)[50];
typedef double (*function)(int *,char);
array (*t[32])(function);
```


Coder et décoder un type C

Il existe un outil nommé `cdecl` que l'on trouve facilement et qui permet de coder en C un type à partir de sa spécification en langue naturelle ou de décoder un type C en langue naturelle.
Essayons :

```
$ cdecl explain "int (*(*t[32])(double (*)(
   int *,char)))[50];"
declare t as array 32 of pointer to function
    (pointer to function (pointer to int,
   char) returning double) returning pointer
    to array 50 of int
$
```

```
$ cdecl declare "var␣as␣pointer␣to␣array␣12␣
   of␣pointer␣to␣array␣54␣of␣pointer␣to␣
   function␣(␣pointer␣to␣void␣)␣returning␣
   pointer␣to␣function␣(␣pointer␣to␣array␣66
   ␣of␣int␣)␣returning␣pointer␣to␣double"
double *(*(*(*var)[12])[54])(void *))(int
   (*)[66])
$
```


Chaîne de production

chaîne de production d'exécutable

Definition (Chaîne de production)

Abusivement appelée chaîne de compilation, la chaîne de production (*toolchain*) est l'ensemble des outils et phases de transformation du **code source** en vue de produire du **code objet** (très souvent un **exécutable**).

exécutable

Definition (Exécutable)

Un **exécutable** (ou **fichier exécutable**) est un fichier contenant (entre autres) les instructions (toutes ou de quoi les avoir toutes) d'un programme, et qui, reconnu comme tel par la plateforme d'exécution peut être directement exécuté sur celle-ci à la demande.

Les principaux formats d'exécutables sont :

- A.OUT (Unix historiques),
- COFF (Unix System V),
- ELF (Unix modernes sauf OSX),
- MACH-O (Darwin-OSX),
- PE (Windows).

```
$ file bonjour
bonjour: Mach-O 64-bit executable x86_64
$ file bonjour.c
bonjour.c: ASCII c program text
$ ./bonjour                          #demande d'exécution
Bonjour tout le monde.
$ ./bonjour.c    #demande d'exécution non satisfaite
zsh: permission denied: ./bonjour.c
$
```


édition de liens

Question

Mais où est définie la fonction `printf`?

Puisqu'elle est appelée, que le programme fonctionne c'est qu'il y a bien quelque part sa suite d'instructions? Un examen complet du code montre que la fonction n'est nulle part définie même si on trouvera bien sa déclaration en examinant le fichier d'inclusion `stdio.h`.

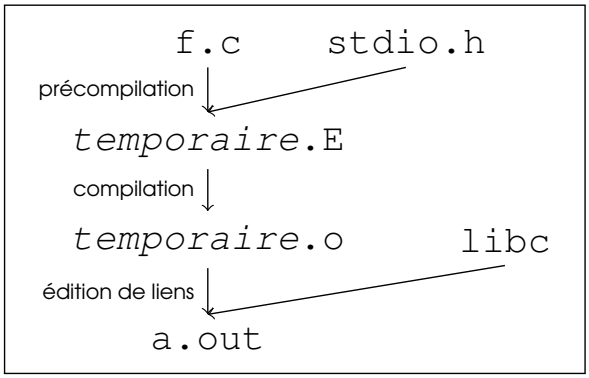
Question

Où est le code de cette fonction et comment est-il intégré à l'exécutable ?

Pour y répondre, il est nécessaire de décomposer la «compilation».

chaîne de production

La commande `gcc f.c` déclenche l'exécution d'une chaîne de compilation comme la suivante:



chaîne de production (simple)

Après exécution de cette chaîne les fichiers *temporaire*`.E` et *temporaire*`.o` sont automatiquement supprimés.

précompilation

Les possibilités offertes par la **précompilation** seront détaillées plus tard, mais il suffit de savoir ici que les directives **`#include`** sont interprétées durant cette phase et permettent d'obtenir l'**inclusion en place du fichier** spécifié.
On peut contrôler la chaîne de compilation `gcc` afin de stopper immédiatement après la précompilation:

```
$ gcc -E monfichier.c
...
 ou
$ gcc -E monfichier.c -o monfichier.e
```

démo

Note

Où l'on voit que l'option `-o` permet de nommer le résultat d'une chaîne de production.

compilation

Definition (Compilation)

La **compilation** consiste à transformer un **code source** en un **code objet**, c'est-à-dire un programme exprimé en langage conçu pour les humains en ce même programme exprimé en langage de la machine machine.
Il s'agit d'une **traduction respectueuse de la structure** du calcul exprimé (pas de déformation qui conduirait à un comportement différent de celui exprimé initialiement).

⚠ Important

Le code (objet) obtenu est incomplet, il ne contient que ce qui a été défini dans le code source de départ. Par exemple, s'il y a un appel à `printf`, la définition de la fonction n'y est normalement pas présente!

On peut contrôler la chaîne `gcc` afin qu'elle stoppe après la production du fichier objet:

```
$ gcc -c monfichier.c
...
 ou
$ gcc -c monfichier.c -o monfichier.o
```

Dans la première forme, le fichier produit est automatiquement nommé du nom de la source substituant l'extension `.o` à l'extension `.c`:

```
$ gcc -c monfichier.c
$ ls monfichier.*
monfichier.c monfichier.o
$
```

La nature du fichier d'extension `.o` peut-être retrouvée:

```
$ file monfichier.o
monfichier.o: Mach-O 64-bit object x86_64
$
```

⚠ Attention

Il ne s'agit pas d'un exécutable! Il contient du code exécutable, mais ne peut-être exécuté car incomplet.

code objet et table des symboles

Un **code objet** n'est pas directement lisible par un humain. Par contre, il existe des outils permettant d'extraire des informations utiles de tels fichiers.
La commande `nm` permet d'obtenir des informations à propos de la **table des symboles** de tels fichiers.

Definition (table des symboles)

C'est une table contenue dans les fichiers objets et qui contient les définitions symboliques nécessaires au bon fonctionnement du programme, c'est-à-dire au moins l'ensemble des **identificateurs à liaison externe** (voir p. 241) déclarés/définis dans le code source.

Exemple de table des symboles

```
#include <stdio.h>

int main() {
  return printf("Coucou, je suis là...\n");
}
```

```
$ gcc -Wall  -c def.c
$ nm def.o
0000000000000000 T _main
                 U _printf
$
```

Ce qui signifie que `main` est un symbole à liaison externe défini dans de code objet et disponible à l'adresse 0 du code (*text*) et que pour son bon fonctionnement `printf` est nécessaire mais non défini (*undefined*).

édition de liens

Il s'agit donc en dernière phase d'obtenir le code de `printf` afin de le lier au code objet obtenu.

Definition (Éditeur de liens)

C'est un outil dont le rôle est de collecter puis agréger différents codes objets de sorte que le **code** soit **clos** (tous les symboles utilisés en partant du `main` sont définis).
On peut utiliser l'éditeur de liens pour fabriquer un code non clos (bibliothèque par exemple), mais son usage de base est d'obtenir un **exécutable**.
Il existe deux **types d'éditions de liens**:

1. **statique**: qui obtient la fermeture lors de l'édition;
2. **dynamique**: qui vérifie que la fermeture est possible, laquelle ne sera obtenue qu'à l'exécution.

liens statiques vs dynamiques

L'**avantage** d'une **liaison statique** est que le code est clos, par conséquent il ne requiert rien de particulier de l'environnement pour s'exécuter.
Les **inconvénients** sont que le code est clos (?!) et qu'il n'est plus possible d'agir sur lui, par exemple en mettant à jour le code avec une nouvelle version de `printf`, il devient nécessaire de repartir ou bien des codes objets ou des codes sources pour refaire a minima une nouvelle édition de liens. Puisque le code est clos, il contient le code de printf, et la place occupée par l'exécutable en est donc plus importante.

Les **avantages** d'une **liaison dynamique** sont que les codes de fonctions des bibliothèques ne sont pas contenus dans l'exécutable, et donc on peut modifier si on le souhaite la bibliothèque à utiliser à l'exécution (mise à jour des bibliothèques transparente), et donc l'espace occupé par l'exécutable en est réduit d'autant. La liaison dynamique va souvent de pair avec le partage des bibliothèques et le système ne contient donc qu'une seule version de printf partagée par tous y compris à l'exécution.
L'**inconvénient** est que les bibliothèques doivent être disponibles à l'exécution, la configuration de l'environnement n'est pas neutre. D'autre part, la liaison dynamique a un coût qui même faible peut parfois impacter la performance.

La chaîne `gcc` permet d'obtenir une **édition de liens statique** avec (exemple sous Linux):

```
$ gcc -static -o def def.o
$ file def
def: ELF 64-bit LSB  executable, x86-64,
   version 1 (GNU/Linux), statically linked,
    for GNU/Linux 2.6.24 BuildID[sha1]=5
   b14dbf1ed51b18bbc49c61f30b34b71ef39399e,
   not stripped
$ nm def
...
000000000004505b0 T fprintf
000000000040105e T main
...
$
```


L' **édition de liens dynamique** est celle obtenue par défaut (exemple sous OSX qui n'autorise plus la liaison statique par défaut):

```
$ gcc -o def def.o
$ file def
def: Mach-O 64-bit executable x86_64
$ nm def
0000000100000000 T __mh_execute_header
0000000100000f50 T _main
                 U _printf
                 U dyld_stub_binder
$ ./def
Coucou, je suis là...
$
```

Note

Les symboles sont toujours non définis mais de nouveaux symboles sont apparus et qui serviront, à l'exécution, à réaliser effectivement la liaison (dynamique donc).

éditeur de liens

Historiquement l'outil d'édition de liens de la chaîne `gcc` est `ld` que l'on peut (plus difficilement) utiliser manuellement.
En utilisant l'option de verbosité `-v` ou `-verbose`, on voit apparaître l'édition de liens réalisée par `ld` sous OSX:

```
$ gcc -v -o def def.o
...
"/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.
     xctoolchain/usr/bin/ld" -demangle -dynamic -arch x86_64 -
     macosx_version_min 10.11.0 -syslibroot /Applications/Xcode.app/
     Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/
     MacOSX10.11.sdk -o def def.o -lSystem /Applications/Xcode.app/
     Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin
     /../lib/clang/7.3.0/lib/darwin/libclang_rt.osx.a
$
```


Sous Linux/Ubuntu, l'outil utliisé est `collect2` (qui lui-même fait appel à `ld`):

```
$ gcc -v -o def def.o
...
/usr/lib/gcc/x86_64-linux-gnu/4.8/collect2 --sysroot=/ --build-id --eh-frame-hdr
      -m elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux
     -x86-64.so.2 -z relro -o def /usr/lib/gcc/x86_64-linux-gnu/4.8/../../../
     x86_64-linux-gnu/crt1.o /usr/lib/gcc/x86_64-linux-gnu/4.8/../../../x86_64-
     linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/4.8/crtbegin.o -L/usr/lib/
     gcc/x86_64-linux-gnu/4.8 -L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../../
     x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../../../lib -L/lib
     /x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../
     lib -L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../.. def.o -lgcc --as-needed -
     lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib
     /gcc/x86_64-linux-gnu/4.8/crtend.o /usr/lib/gcc/x86_64-linux-gnu
     /4.8/../../../x86_64-linux-gnu/crtn.o
$
```

⚠ Attention

L'édition de liens manuelle est une opération subtile très dépendante de la plate-forme et de la chaîne de compilation.

Avant d'illustrer la construction modulaire de code, intéressons-nous à la précompilation (utile à maîtriser aussi pour la modularité)...

précompilation

Par défaut la première phase de la chaîne de production est une phase de **précompilation**. Elle est propre au langage et C (et C++).
En effet, les codes source C sont généralement impurs, car ils contiennent des constructions qui ne font pas partie du langage C proprement dit comme les macro-définitions, des conditionnelles, des inclusions ou encore des directives spécifiques.
Cette phase est très pauvre en analyse statique du code, il s'agit essentiellement d'effectuer des substitutions.

Definition (Précompilation – preprocessing)

C'est une phase de la chaîne de production du C qui permet de transformer un code C impur en code C pur en:

- substituant les macro-définitions par leur expansion (**`#define`**),
- excluant certaines parties du code si indiqué (**`#if`**...),
- incluant du code C contenu dans d'autres fichiers source (**`#include`**).

mais aussi en modifiant le comportement de la chaîne en suivant certaines directives (**`#pragma`**...).

Les directives de précompilation forment un (mini) langage de contrôle qui permet entre autre d'obtenir un paramétrage plus aisé des codes sources comme la dépendance vis-à-vis de caractéristiques de la plate-forme par exemple.
Les directives doivent être placées une par ligne et seules sur leur ligne.
Les directives standard sont :

- **`#if`**, **`#ifdef`**, **`#ifndef`**, **`#elif`**, **`#else`**, **`#endif`**,
- **`#include`**,
- **`#define`**, **`#undef`**,
- **`#line`**, **`#error`**,
- **`#pragma`**.

Certaines correspondent à des conditionnelles pour lesquelles le mot-clé `defined` est utilisable.

#include

Definition (#include)

permet d'obtenir le remplacement en lieu et place de cette directive par le contenu du fichier correspondant à la référence spécifiée:

```
#include <référence>
#include "référence"
#include symbole
```


La **première forme** permet d'inclure le fichier standard de référence donnée et dont l'emplacement est défini par l'environnement du compilateur (en général dans des répertoires standards type `/usr/include`).
La **seconde forme** permet d'inclure le fichier de référence donnée dont l'emplacement est défini par l'environnement (par défaut le répertoire courant). Si la référence ne correspond à rien, la directive est alors traitée comme étant de la première forme.
La **troisième forme** est la forme paramétrée, puisque le symbole peut-être défini auparavant par une directive **`#define`**.

La directive sert en premier lieu à inclure des fichiers dits d'en-tête contenant essentiellement des prototypes de fonctions et des types.

Exemple

L'utilisation de la fonction `printf` nécessite son prototype qui se trouve dans l'en-tête standard `stdio.h`.

```
#include <stdio.h>

int main() {
  printf("Coucou, je suis là...\n");
}
```



```
$ gcc -Wall inc.c -o inc
$ ./inc
Coucou, je suis là...
$
```

Mauvais exemple

```
int main() {
  printf("Coucou, je suis là...\n");
}
```

```
$ gcc -Wall incbad.c -o incbad
incbad.c:2:3: warning: implicitly declaring library
    function 'printf' with type 'int (const char *,
    ...)' [-Wimplicit-function-declaration]
  printf("Coucou, je suis là...\n");
  ^
incbad.c:2:3: note: include the header <stdio.h> or
    explicitly provide a declaration for 'printf'
1 warning generated.
$ ./incbad
Coucou, je suis là...
$
```


#define

Definition (#define)

permet d'obtenir le remplacement, dans la suite du code source, d'un symbole (éventuellement paramétré) par un motif de substitution.

```
#define symbole motif
#define symbole(liste) motif
```

La *liste* de substitution peut être terminée par `...` afin obtenir une liste d'arguments de longueur variable.
Le *motif*, lui, peut contenir des opérateurs `#` ou `##`.

Exemple

```
#include <stdio.h>

#define MSG "Hello"

int main(void) {
  printf("%s\n",MSG);
}
```

dont le code C pur produit est :

```
...
int main(void) {
  printf("%s\n","Hello");
}
```


Il est fréquent de définir des «constantes» à l'aide de **#define** :

Exemple

```
#include <stdio.h>

#define N 100

void f(int *t) {
  for (int i=0; i<N; i++) printf("%d␣",t[i]);
  putchar('\n');
}

int main(void) {
  int t[N];
  for (int i=0; i<N; i++) t[i] = i*i;
  f(t);
}
```

Attention

La forme avec *liste* peut-être vue comme une fonction, mais attention, il ne faut pas s'illusionner, il s'agit d'une simple substitution syntaxique! il n'y a d'ailleurs **aucune** vérification de type. Cela peut poser de très sérieux problèmes... C'est pourquoi leur manipulation intensive requiert beaucoup de délicatesse!

Exemple

```
#include <stdio.h>

#define ECRETE(x) x>10?10:x

int main(void) {
  for (int i=0; i<13; i++)
    printf("%d ",ECRETE(i));
  putchar('\n');
}
```


L'exécution produit :

```
$ ./defi2
0 1 2 3 4 5 6 7 8 9 10 10 10
$
```

Le code C pur produit est :

```
...
int main(void) {
  for (int i=0; i<13; i++)
    printf("%d ",i>10?10:i);
  putchar('\n');
}
```

⚠ Attention

Le «piège» est que les arguments ne sont pas évalués.
Il ne sont que l'objet d'une substitution...

Exemple piège

```
#include <stdio.h>

#define ECRETE(x) x>10?10:x

int main(void) {
  for (int i=0; i<13;)
    printf("%d␣",ECRETE(i++));
  putchar('\n');
}
```



```
$ ./defi3
1 3 5 7 9 11 10
$
```

C pur :

```
...
int main(void) {
  for (int i=0; i<13;)
    printf("%d␣",i++>10?10:i++);
  putchar('\n');
}
```

Autre piège (classique)

```
#include <stdio.h>

#define MULT(x,y) x*y

int main(void) {
  printf("%d\n",MULT(3,4));
  printf("%d\n",MULT(2+1,1+3));
}
```

```
$ ./defi4
12
6
$
```


La substitution en cascade est naturelle (les arguments sont étendus avant :

Exemple (toujours mauvais)

```
#include <stdio.h>

#define ADD(x,y) x+y
#define MULT(x,y) x*y

int main(void) {
  printf("%d\n",MULT(3,4));
  printf("%d\n",MULT(ADD(2,1),ADD(1,3)));
}
```

C pur :

```
...
int main(void) {
  printf("%d\n",3*4);
  printf("%d\n",2 +1*1 +3);
}
```

opérateur

L'opérateur **#**, employé dans le motif de remplacement en préfixe d'un argument permet d'obtenir, non le remplacement de l'argument mais la chaîne de caractères correspondant à l'argument :

Exemple

```
#include <stdio.h>

#define DBG_CALL(f,x) printf("calling␣%s\n",#f "(" #x
     ")"); f(x)

void f(int i) {}
void g(int i) {}

int main(void) {
  DBG_CALL(f,4);
  DBG_CALL(g,5);
}
```



```
$ ./defi6
calling f(4)
calling g(5)
$
```

L'opérateur

L'opérateur **##** permet d'obtenir la concaténation symbolique de deux arguments (ou plus) :

Exemple

```
#include <stdio.h>

#define VERSION_CALL(f,v,x) f##v(x)

void f1(int i) { printf("f1\n"); }
void f2(int i) { printf("f2\n"); }

int main(void) {
  VERSION_CALL(f,1,4);
  VERSION_CALL(f,2,4);
}
```



```
$ ./defi7
f1
f2
$
```

Exemple

```
#include <stdio.h>

#define VERSION 1

#define TRICK2(f,v,x) f##v(x)
#define TRICK1(f,v,x) TRICK2(f,v,x)
#define VERSION_CALL(f,x) TRICK1(f,VERSION,x)

void f1(int i) { printf("f1\n"); }
void f2(int i) { printf("f2\n"); }

int main(void) {
  VERSION_CALL(f,4);
  VERSION_CALL(f,4);
}
```



```
$ ./defi8
f1
f1
$
```

redéfinition

Attention

La redéfinition d'un symbole n'est autorisée que dans certains cas très précis; il vaut mieux éviter de le faire (voir p. 508).

Interdit

```
#define ARGH 666
#define ARGH 777
```


définition en ligne de commande

Les macro-définitions peuvent aussi être fournies en ligne de commande. Soit le code :

Exemple

```
#include <stdio.h>

#define TRICK2(f,v,x) f##v(x)
#define TRICK1(f,v,x) TRICK2(f,v,x)
#define VERSION_CALL(f,x) TRICK1(f,VERSION,x)

void f1(int i) { printf("f1\n"); }
void f2(int i) { printf("f2\n"); }

int main(void) {
  VERSION_CALL(f,4);
  VERSION_CALL(f,4);
}
```

Dont la compilation (et l'exécution) peut être obtenue par :

```
$ gcc -DVERSION=2  defi9.c   -o defi9  # définition du symbole VERSION!!!
$ ./defi9
f2
f2
$
```

Note

Il est rare de définir de façon externe autre chose que des constantes (même si tout est possible...).

#undef

Definition (#undef)

Cette directive permet à partir de ce point de faire disparaître la précédente définition du symbole s'il existait et rien sinon, ainsi pour redéfinir un symbole il vaut mieux écrire :

Exemple

```
#define VERSION 1
#undef VERSION
#define VERSION 2
#undef TOTO
```

inclusion conditionnelle

Definition (inclusion conditionnelle)

L'inclusion conditionnelle permet de contrôler quelle portion du code source ou non sera prise en compte, ce fonction de la valeur d'expressions constantes. Parmi les expressions `defined` permet de tester l'existence d'un symbole du préprocesseur.

`#if, #elif, #else, #endif`

La syntaxe est :

```
#if expression
#endif

#if expression
#else
#endif

#if expression
#elif expression
#elif expression
...
#endif
```

Exemple

```
#include <stdio.h>

#define DEBUG_LEVEL  0

void f(int i) {
#if DEBUG_LEVEL == 1
  printf("call to f()\n");
#elif DEBUG_LEVEL == 2
  printf("call to f(%d)\n",i);
#endif
}

int main(void) {
  f(5);
}
```


#ifdef, #ifndef

Definition (#ifdef, #ifndef)

Ces deux opérateurs sont respectivement équivalents à **#if** `defined` et **#if** `!defined`.

#line

Definition (#line)

Cette directive permet de contrôler le numéro de ligne du code source ainsi que le nom présumé du fichier courant. L'effet n'est visible qu'en cas d'avertissement ou d'erreur.
Ce n'est pas une directive fréquemment utilisé par les programmeurs, mais plutôt par les outils générateurs de code.

Exemple

```
#include <stdio.h>

#line 666 "factice.c"
fdsjkflds;

int main(void) {
#line 3 "bidon.c"
  f(5);
}
```

```
$ make ./line
factice.c:666:1: warning: type specifier missing, defaults to 'int'
      [-Wimplicit-int]
fdsjkflds;
^
bidon.c:3:3: warning: implicit declaration of function 'f' is invalid
      in C99
      [-Wimplicit-function-declaration]
  f(5);
  ^
2 warnings generated.
...
$
```

#error

Definition (#error)

Cette directive permet de générer une erreur de compilation.
Elle permet par exemple d'arrêter la compilation si des définitions de symboles ne sont pas corrects

Exemple

```
#include <stdio.h>

#ifndef VERSION
#error "Qu'est ce que tu fiches ?"
#endif

int main(void) {
}
```

```
$ make error
cc --std=c11 -pedantic -Wall    error.c   -o error
error.c:4:2: error: "Qu'est ce que tu fiches ?"
#error "Qu'est ce que tu fiches ?"
 ^
1 error generated.
make: *** [error] Error 1
$
```

Exemple

```
#include <stdio.h>

#ifndef VERSION
#error "Need to define VERSION"
#elif VERSION<1 || VERSION>2
#error "Bad VERSION (1 or 2)"
#endif

#define TRICK2(f,v,x) f##v(x)
#define TRICK1(f,v,x) TRICK2(f,v,x)
#define VERSION_CALL(f,x) TRICK1(f,VERSION,x)

void f1(int i) { printf("f1\n"); }
void f2(int i) { printf("f2\n"); }

int main(void) {
  VERSION_CALL(f,4);
  VERSION_CALL(f,4);
}
```


inclusions mutliples

Un exemple classique d'utilisation des **#define**, **#ifdef** est la protection contre les inclusions multiples (qui peuvent être gênantes) :

Exemple

```
/*
 * inc.h
 */
struct maStruct {
  int i;
};
```

```
#include "inc.h"
#include "inc.h"

int main(void) {
}
```

La solution est :

Exemple

```
/*
 * myinc.h
 */
#ifndef MYINC_H
#define MYINC_H
struct maStruct {
  int i;
};
#endif
```

```
#include "myinc.h"
#include "myinc.h"

int main(void) {
}
```


symboles prédéfinis

On ne les citera pas tous, mais le standard définit un certain nombre de symboles :

- `__DATE__` la date de compilation;
- `__FILE__` le nom du fichier courant;
- `__LINE__` le numéro de la ligne;
- `__STDC__` qui vaut 1 si le compilateur est conforme, 0 sinon;
- `__STDC_HOSTED__` qui vaut 1 si l'environnement est celui d'un système d'exploitation, 0 sinon;
- `__STDC_VERSION__` qui vaut le numéro du standard;
- `__TIME__` qui vaut l'heure de compilation.

Exemple

```
#include <stdio.h>

int main(void) {
#ifndef __STDC__
#error "Oulala ce compilo est vraiment zarbi"
#endif
#if __STDC__ == 1
  printf("J'ai été compilé le %s à %s\n",__DATE__,
      __TIME__);
  printf("Mon compilo est conforme à la norme %ld\n",
      __STDC_VERSION__);
#ifdef __STDC_HOSTED__
  printf("On est sur un système d'exploitation\n");
#endif
#endif
  printf("source %s ligne %d\n",__FILE__,__LINE__);
}
```

```
$ ./symb
J'ai été compilé le Sep 27 2016 à 17:25:14
Mon compilo est conforme à la norme 201112
On est sur un système d'exploitation
source symb.c ligne 14
$
```


modularité

Definition (Modularité)

Technique de conception consistant à **regrouper des fonctionnalités en ensembles logiciels cohérents, indépendants et interchangeables**. Ceci afin d'obtenir une cohérence structurelle ou fonctionnelle, une séparation des responsabilités, un couplage faible et de l'abstraction.
En langage C, la modularité s'obtient *via*:

1. le découpage du code source ou,
2. l'utilisation de bibliothèques.

Par exemple, module de connexion à la base de donnée, module de communication réseau, module de gestion de l'interface utilisateur...

bibliothèque

Definition (Bibliothèque)

Module réutilisable dont les fonctionnalités sont d'un **usage général** et qui transcendent celles d'une application particulière.
La bibliothèque est un ensemble logiciel dans lequel on vient piocher ce qui est nécessaire.

Par exemple, bibliothèque de fonctions mathématiques, d'entrées/sorties, interface graphique...

modularité par découpage

```
#ifndef ES_MODULE
#define ES_EXTERN extern
#else
#define ES_EXTERN
#endif

ES_EXTERN int lire(int *data_to_read);
ES_EXTERN void ecrire(int data_to_write);
```

```
#include "es.h"

int main() {
  int valeur;
  while (lire(&valeur)!=0) {
    ecrire(valeur);
  }
}
```

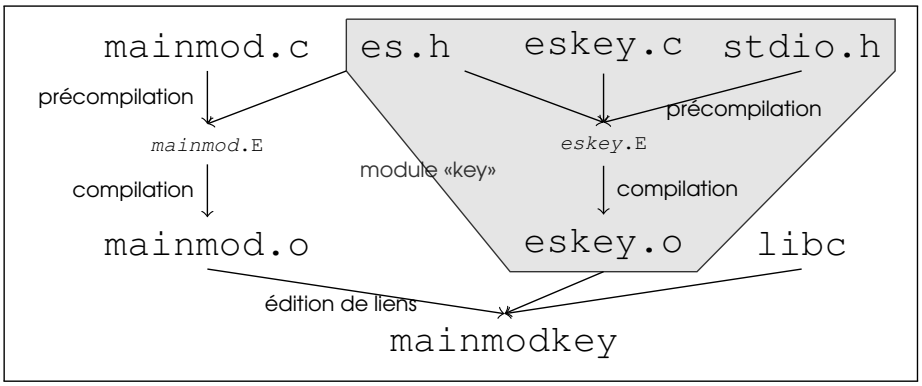
```
#define ES_MODULE
#include <stdio.h>
#include "es.h"

int lire(int *v) {
  puts("Give a number");
  return scanf("%d",v)!=1?0:1;
}

void ecrire(int v) {
  printf("Read: %d\n",v);
}
```

Ces différents fichiers de code source peuvent être compilés séparément puis liés:

```
$ gcc -c mainmod.c
$ gcc -Wall -c mainmod.c
$ gcc -Wall -c eskey.c
$ gcc -o mainmodkey mainmod.o eskey.o
$ ./mainmodkey
Give a number
12
Read: 12
Give a number
43
Read: 43
Give a number
$
```


modularité et chaîne de production

La chaîne correspondante est la suivante:



Chaîne de production modulaire

modularité par découpage

On peut remplacer le module d'entrées/sorties «clavier» par un autre (qui lit dans un fichier par exemple):

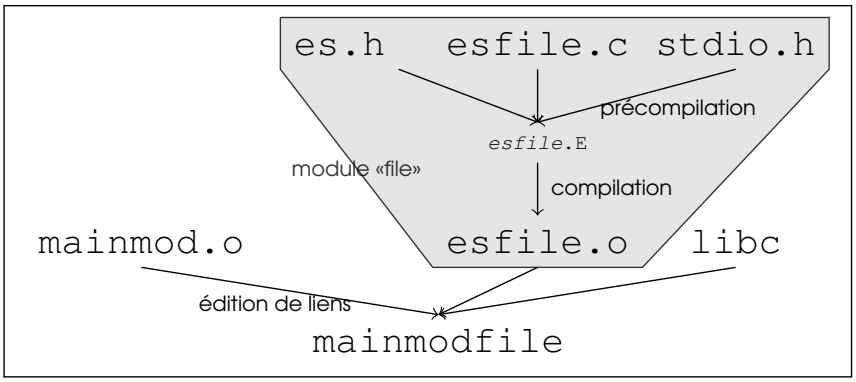
```
#define ES_MODULE
#include <stdio.h>
#include <stdlib.h>
#include "es.h"

static FILE *file=NULL;

static void ouvre() {
  file = fopen("data.txt","r");
  if (file==NULL) fprintf(stderr,"failure\n"), exit(1);
}

int lire(int *v) {
  if (file==NULL) ouvre();
  if (fscanf(file,"%d",v)!=1) { fclose(file); return 0; }
  return 1;
}

void ecrire(int v) {
  printf("Read: %d\n",v);
}
```


En supposant que les compilations précédentes aient été réalisées (et qu'un fichier d'entrées soit disponible):

```
$ gcc -Wall -c esfile.c
$ gcc -o mainmodfile mainmod.o esfile.o
$ ./mainmodfile
Read: 123
Read: 666
Read: 999
Read: 42
$
```

modularité et chaîne de production

La chaîne correspondante est la suivante:



Chaîne de production modulaire

modularité par découpage

Note

Nous avons ainsi obtenu deux exécutables:

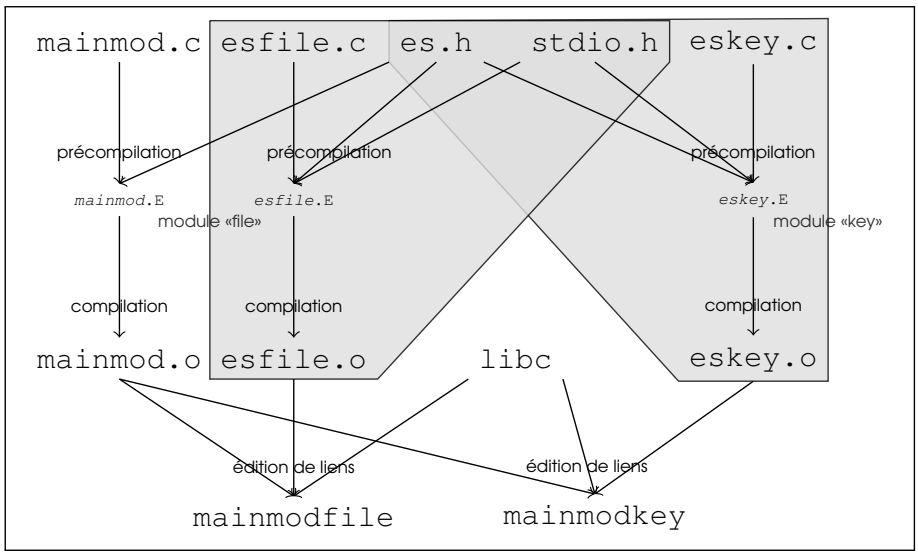
- à partir d'un seul **main** (c'est un exemple),
- lié à différents modules.

Le fichier d'en-têtes `es.h` est un **contrat** entre les deux parties:

1. le programme qui s'engage à utiliser les fonctionnalités telles que déclarées,
2. et le module qui s'engage à définir les fonctionnalités telles que déclarées.

modularité et chaîne de production

La chaîne correspondante est la suivante:



Chaîne de production modulaire

`make` et `makefile`

On peut constater que dans le cadre d'une modularité, même simple, le contrôle manuel de la chaîne de compilation est au moins casse-pieds, sinon délicat.

Definition (`make`)

L'outil `make` permet de spécifier un arbre de dépendance des codes sources et d'indiquer quelles actions sont à réaliser lorsqu'une dépendance est modifiée.
La spécification prend place dans des fichiers dits `makefile`.

Le fonctionnement de `make` repose sur l'examen des dates de dernière modification des fichiers concernés, de sorte que si la cible est plus vieille que l'une des sources, c'est qu'il faut regénérer la cible à partir des sources (possiblement à l'aide des actions spécifiées par la règle).

La spécification *via* un `makefile` de la chaîne de production de l'exemple précédent est (possiblement) :

Exemple

```
eskey.o : eskey.c es.h
        gcc --std=c11 -pedantic -Wall -c eskey.c

esfile.o : esfile.c es.h
        gcc --std=c11 -pedantic -Wall -c esfile.c

mainmod.o : mainmod.c es.h
        gcc --std=c11 -pedantic -Wall -c mainmod.c

mainmodfile : mainmod.o esfile.o
        gcc -o mainmodfile mainmod.o esfile.o

mainmodkey : mainmod.o eskey.o
        gcc -o mainmodkey mainmod.o eskey.o
```

L'outil `make` peut être invoqué de la façon suivante :

```
$ make -f Makefile.mainmod mainmodfile
gcc --std=c11 -pedantic -Wall -c mainmod.c
gcc --std=c11 -pedantic -Wall -c esfile.c
gcc -o mainmodfile mainmod.o esfile.o
$ make -f Makefile.mainmod mainmodfile
$
```

On voir alors la chaîne de production s'exécuter en conséquence. La seconde fois rien ne se produit car l'exécutable existe déjà (voir suite)...

dépendance de `makefile`

Definition (dépendance)

La spécification d'une dépendance est de la forme :

cible : *source*...

et permet d'indiquer que le **fichier** *cible* dépend des **fichiers** *source*s spécifiés.
L'outil `make` utilise une telle spécification en vérifiant :

- soit que la *cible* n'existe pas;
- soit que la *cible* est plus vieille qu'au moins une des sources.

auquel cas, si une action correspondante existe, elle est réalisée.

action de `makefile`

Definition (action)

La spécification d'une action est de la forme :

commande
...

où toute commande peut être utilisée. Si une commande échoue, la chaîne est interrompue et un diagnostic est affiché.

Attention

Toute action doit être spécifiée en commençant la ligne par un caractère de tabulation (pas d'espaces! une tabulation).

invocation de `make`

Definition (invocation)

L'invocation de l'outil `make` a la forme :

`make` [`-f` *makefile*] [*cible*...]

qui déclenche la remontée de l'arbre à partir de la cible et le déclenchement en sens inverse des actions associées si nécessaire.

Note

Si on ne spécifie pas de *cible*, c'est la première rencontrée qui est utilisée.

macros de `make`

Pour simplifier l'écriture de `makefile`s il existe la possibilité de définir des macros :

Definition (macros)

Une macro est la définition d'un symbole associée à une valeur :

SYMBOLE=valeur

Une convention est d'orthographier les macros avec des caractères majuscules.
Les macros peuvent alors être étendues (remplacées par leur valeur) *via* la syntaxe :

`$(`*SYMBOLE*`)`

Une simplification du makefile précédent peut être obtenue ainsi :

Exemple

```
CC = gcc
CFLAGS = --std=c11 -pedantic -Wall
COMPILE = $(CC) $(CFLAGS)

eskey.o : eskey.c es.h
        $(COMPILE) -c eskey.c

esfile.o : esfile.c es.h
        $(COMPILE) -c esfile.c

mainmod.o : mainmod.c es.h
        $(COMPILE) -c mainmod.c

mainmodfile : mainmod.o esfile.o
        $(CC) -o mainmodfile mainmod.o esfile.o

mainmodkey : mainmod.o eskey.o
        $(CC) -o mainmodkey mainmod.o eskey.o
```

fausse cible

Il est fréquent de vouloir obtenir la réalisation d'actions pour des cibles qui ne sont pas des fichiers. Par exemple une cible pour «packager» le logiciel, nettoyer, etc.

Definition (`.PHONY`)

Cette cible prédéfinie permet de déclarer des cibles qui ne correspondent pas à des fichiers, mais permet de déclencher des actions.

Exemple

```
CC = gcc
CFLAGS = --std=c11 -pedantic -Wall
COMPILE = $(CC) $(CFLAGS)

.PHONY : all clean

eskey.o : eskey.c es.h
        $(COMPILE) -c eskey.c

esfile.o : esfile.c es.h
        $(COMPILE) -c esfile.c

mainmod.o : mainmod.c es.h
        $(COMPILE) -c mainmod.c

mainmodfile : mainmod.o esfile.o
        $(CC) -o mainmodfile mainmod.o esfile.o

mainmodkey : mainmod.o eskey.o
        $(CC) -o mainmodkey mainmod.o eskey.o

all : mainmodkey mainmodfile

clean :
        rm mainmodfile mainmodkey eskey.o esfile.o mainmod.o
```

macros prédéfinies

Definition (macros prédéfinies)

Dans la spécification d'une action il est possible de faire référence symboliquement à certains des éléments de la règle de dépendance :

- `$@` désigne la cible,
- `$?` désigne les sources plus récentes que la cible,
- `$<` désigne la première source de la règle,
- `$^` désigne l'ensemble des sources de la règle.

Exemple

```
CC = gcc
CFLAGS = --std=c11 -pedantic -Wall
LDFLAGS =
COMPILE = $(CC) $(CFLAGS)
LINK = $(CC) $(LDFLAGS)

.PHONY : all clean
.SILENT : all clean

eskey.o : eskey.c es.h
        $(COMPILE) -c $<

esfile.o : esfile.c es.h
        $(COMPILE) -c $<

mainmod.o : mainmod.c es.h
        $(COMPILE) -c $<

mainmodfile : mainmod.o esfile.o
        $(CC) -o $@ $^

mainmodkey : mainmod.o eskey.o
        $(CC) -o $@ $^

all : mainmodkey mainmodfile
        echo "Generated: $^"

clean :
        echo "Cleaning..."
        -rm mainmodfile mainmodkey eskey.o esfile.o mainmod.o
```

localisation des includes `-I`

Il n'est pas rare de séparer les fichiers de différentes natures dans des répertoires séparés. Par exemple, `src` pour les codes sources C, `include` pour les en-têtes, etc.
On peut toujours réaliser les **`#include`** en conséquence, mais cela interdit toute souplesse. `gcc` permet de contrôler la résolution de la localisation des fichiers inclus *via* :

Definition (`-I`)

Cette option dont la syntaxe est `-I répertoire` permet d'ajouter le répertoire spécifié à la liste des répertoires dans lesquels sont recherchés les fichiers inclus.

Si l'arbre des sources est découpé ainsi :

```
$ tree --charset ascii
.
|-- Makefile
|-- include
|   `-- es.h
`-- src
    |-- esfile.c
    |-- eskey.c
    `-- mainmod.c

2 directories, 5 files
$
```

et que l'on souhaite générer les fichiers construits dans le répertoire `build`, le makefile suivant fera l'affaire.

Exemple

```
INC = include
SRC = src
BUILD = build
CC = gcc
CFLAGS = --std=c11 -pedantic -Wall -I$(INC)
LDFLAGS =
COMPILE = $(CC) $(CFLAGS)
LINK = $(CC) $(LDFLAGS)

.PHONY : all clean
.SILENT : all clean

$(BUILD)/eskey.o : $(SRC)/eskey.c $(INC)/es.h
        @test -d $(BUILD) || mkdir $(BUILD)
        $(COMPILE) -c $< -o $@

$(BUILD)/esfile.o : $(SRC)/esfile.c $(INC)/es.h
        @test -d $(BUILD) || mkdir $(BUILD)
        $(COMPILE) -c $< -o $@

$(BUILD)/mainmod.o : $(SRC)/mainmod.c $(INC)/es.h
        @test -d $(BUILD) || mkdir $(BUILD)
        $(COMPILE) -c $< -o $@

$(BUILD)/mainmodfile : $(BUILD)/mainmod.o $(BUILD)/esfile.o
        @test -d $(BUILD) || mkdir $(BUILD)
        $(CC) -o $@ $^

$(BUILD)/mainmodkey : $(BUILD)/mainmod.o $(BUILD)/eskey.o
        @test -d $(BUILD) || mkdir $(BUILD)
        $(CC) -o $@ $^

all : $(BUILD)/mainmodkey $(BUILD)/mainmodfile
        echo "Generated: $^"

clean :
        echo "Cleaning..."
        rm -rf build
```



```
$ make all
gcc --std=c11 -pedantic -Wall -Iinclude -c src/mainmod.c -o
    build/mainmod.o
gcc --std=c11 -pedantic -Wall -Iinclude -c src/eskey.c -o
    build/eskey.o
gcc -o build/mainmodkey build/mainmod.o build/eskey.o
gcc --std=c11 -pedantic -Wall -Iinclude -c src/esfile.c -o
    build/esfile.o
gcc -o build/mainmodfile build/mainmod.o build/esfile.o
Generated: build/mainmodkey build/mainmodfile
$ tree --charset ascii
.
|-- Makefile
|-- build
|   |-- esfile.o
|   |-- eskey.o
|   |-- mainmod.o
|   |-- mainmodfile
|   `-- mainmodkey
|-- include
|   `-- es.h
`-- src
    |-- esfile.c
    |-- eskey.c
    `-- mainmod.c

3 directories, 10 files
$
```

édition de lien avec une bibliothèque

Definition (`-l`)

Cette option dont la syntaxe est `-l`*nom* permet d'obtenir l'édition de liens avec une bibliothèque dont le nom est `lib`*nom*`.`*extension*, où *extension* est soit :

- celle d'une bibliothèque dynamique (selon les systèmes `so`, `dylib` pour OSX)
- celle d'une bibliothèque statique (en général `a`).

Note

La localisation des bibliothèque est propre à chaque système, mais en général elles sont recherchées dans `/usr/lib`, `/usr/lib/gcc/...`.

localisation des bibliothèques

Definition (`-L`)

Cette option dont la syntaxe est `-L` *repertoire* permet d'ajouter le répertoire spécifié dans le chemin de recherche des bibliothèques à l'édition de liens.