



UNIVERSITÉ  
CÔTE D'AZUR

# Programmation impérative en C

## Cours 6. Modularité et compilation

---

Olivier Baldellon

Courriel : `prénom.nom@univ-cotedazur.fr`

Page professionnelle : `https://upinfo.univ-cotedazur.fr/~obaldellon/`

LICENCE 2 — FACULTÉ DES SCIENCES ET INGÉNIERIE DE NICE — UNIVERSITÉ CÔTE D'AZUR

- ▶ Rappel :
  - ▶ Partiel la semaine prochaine
  - ▶ Pas de TP la semaine prochaine
  
- ▶ Suite du cours :
  - ▶ On a tout vu sur le langage C : maintenant il faut digérer et appliquer
  - ▶ On commencera dans 15 jours un TP filé
  - ▶ Cinq séances pour créer un logiciel
  - ▶ Il y aura toujours des CM (présentation du TP, discussion des solutions).

- 🍃 Partie I. Le préprocesseur
- 🍃 Partie II. Usage avancé préprocesseur
- 🍃 Partie III. Fichier Headers
- 🍃 Partie IV. Compilation séparée
- 🍃 Partie V. La bibliothèque standard (ANSI)
- 🍃 Partie VI. Autres bibliothèques
- 🍃 Partie VII. Table des matières

- ▶ J'ai une promo de 130 étudiants.
  - ▶ Je n'ai pas envie de mettre « 130 » en paramètre dans chaque fonction.
  - ▶ Mais ce n'est pas très propre de coder cette valeur en dur.

```
#include <stdio.h>
/* Par défaut, tout le monde à 12/20,
   c'est plus rapide que de corriger */
void remplir(int notes[]) {
    int i;
    for (i=0, i<130, i++) { notes[i] = 12; }
}

int moyenne(int notes[]) {
    int i, somme = 0;
    for (i=0; i<130; i++) { somme += notes[i]; }
    return somme/130;
}

int main(void) {
    int notes[130]; /* impossible de remplacer 130 par une variable*/
    int m;
    remplir(notes);
    m = moyenne(notes);
    printf("La moyenne est %d\n",m);
    return 0;
}
```

\*.C

- ▶ Le préprocesseur remplace un identifiant par une valeur.
  - ▶ cela se passe avant la compilation avec la directive `#define`
  - ▶ ce n'est pas une variable!

```
#include <stdio.h>
#define NOMBRE_ETUDIANT 130
/* Par défaut, tout le monde à 12/20, plus rapide que de corriger */
void remplir(int notes[]) {
    int i;
    for (i=0, i<NOMBRE_ETUDIANT, i++) notes[i] = 12;
}

int moyenne(int notes[]) {
    int i, somme = 0;
    for (i=0; i<NOMBRE_ETUDIANT; i++) somme += notes[i];
    return somme/NOMBRE_ETUDIANT;
}

int main(void) {
    int notes[NOMBRE_ETUDIANT]; int m;
    remplir(notes);
    m = moyenne(notes);
    printf("La moyenne est %d\n",m);
    return 0;
}
```

\*.C

- ▶ On peut mettre définir des macros avec des arguments.

- La macro suivante

```
#define MIN(A,B) ((A)>(B) ? (B) : (A))
```

```
*.c
```

- remplacera **avant la compilation** la ligne

```
note = MIN(note+2,20);
```

```
*.c
```

- par :

```
note = ((note+2)>(20) ? (20) : (note+2));
```

```
*.c
```

- ▶ La macro ne calcule rien, mais remplace du texte par du texte.

- ▶ Les macros ne sont pas des fonctions

```
/* Une macro */  
#define MIN(A,B) ((A)>(B) ? (B) : (A))  
  
/* Une fonction */  
int min(int a, int b) { return a>b ? b : a ;}
```

\*.c

- Comparez

```
int i=10; int j=10;  
int k0 = min(i++,j++);
```

\*.c

- avec

```
int i=10; int j=10;  
int k1 = MIN(i++,j++);
```

\*.c

- ▶ Que vaut k0 et k1 ? et i et j ?

- ▶ Considérons une macro qui calcule le carré d'un nombre :

```
#define CARRE(X) = X*X /* Où est l'erreur ? */
```

\*.C

- ▶ Appliquons cette macro à l'exemple suivant :

```
int x = 3  
x = CARRE(x+1,x+1)
```

\*.C

- ▶ Que se passe-t-il ? Comment le corriger ?

```
#define CARRE(X) = ((X)*(X)) /* Version correcte */
```

\*.C

- ▶ Attention à ne pas oublier les parenthèses !
  - ▶ Une macro remplace du texte par du texte.
  - ▶ Une macro ne comprend pas le C.



- 🍃 Partie I. Le préprocesseur
- 🍃 Partie II. Usage avancé préprocesseur
- 🍃 Partie III. Fichier Headers
- 🍃 Partie IV. Compilation séparée
- 🍃 Partie V. La bibliothèque standard (ANSI)
- 🍃 Partie VI. Autres bibliothèques
- 🍃 Partie VII. Table des matières

- ▶ Le compilateur gcc possède de nombreuses options, parmi lesquelles
  - **-E** n'exécute que le préprocesseur.
    - ▶ et affiche le résultat sur la sortie standard
  - Certaines options permettent d'optimiser.
    - ▶ `-O1`, `-O2` ou `-O3` pour optimiser
    - ▶ `-O0` pour ne pas optimiser (par défaut)
    - ▶ L'optimisation peut donner des comportement étrange dans les *Undefined behaviour*.
  - Définir et supprimer une macro : **-D** et **-U**
    - ▶ `-DMA_MACRO=quelquechose` : crée la macro `MA_MACRO` avec comme valeur `quelquechose`
    - ▶ `-DMA_MACRO` : crée la macro `MA_MACRO` avec une valeur vide par défaut
    - ▶ `-UMA_MACRO=quelquechose` : supprime la macro `MA_MACRO` qui devient non définie.

```
#include <stdio.h>
#define DEBUG(X) printf(#X " = %d\n",X)

int main(void) {
    int x=3;
    int y=2;
    y++;
    DEBUG(y);
    x=x+1;
    DEBUG(x+1);
    return 0;
}
```

\*.c

```
y=3
x+1=5
```

SHELL

► Explication :

- Un # devant le nom de l'argument le transforme en chaîne de caractère.
- Deux chaînes (écrite avec guillemets ") consécutives sont concaténées
- Impossible à faire avec une fonction

```
DEBUG(y);           /* Dans le code source */
printf("y" " = %d\n",y); /* Après passage du préprocesseur */
printf("y = %d\n",y);  /* puis interprétation du compilateur*/
```

\*.c

```
#include <stdio.h>
/*#define MODE_DEBUG*/
#ifdef MODE_DEBUG /* Si MODE_DEBUG est définie */
#define DEBUG(X) printf(#X " = %d\n",X)
#else
#define DEBUG(X)
#endif

int main(void) {
    int x=3;
    x=x+1;
    DEBUG(x+1);
    printf("ok\n");
    return 0;
}
```

\*.C

- ▶ Si on décommente la ligne `#define MODE_DEBUG`

```
x+1=5
ok
```

SHELL

- ▶ Si on la laisse commentée : `/*#define MODE_DEBUG*/`
  - ▶ `MODE_DEBUG` n'est pas définie
  - ▶ `DEBUG` est la macro vide

ok

SHELL

- ▶ Par défaut la variable `DEBUG_OPTIONS` est vide
- ▶ En appelant la règle `debug` je change cette variable.

```
CC = gcc
OPTIONS = -Wall -ansi -pedantic
DEBUG_OPTIONS =

debug: DEBUG_OPTIONS = -DMODE_DEBUG
debug: programme

all: programme

%: %.c
    $(CC) $(OPTIONS) $(DEBUG_OPTIONS) $< -o $@
```

Makefile

```
olivier@valrose:~ $ make
gcc -Wall -ansi -pedantic programme.c -o programme
olivier@valrose:~ $ ./programme
ok
```

SHELL

```
olivier@valrose:~ $ make debug
gcc -Wall -ansi -pedantic -DMODE_DEBUG programme.c -o programme
olivier@valrose:~ $ ./programme
x+1=5
ok
```

SHELL

- ▶ Il existe plusieurs macro prédéfinies :
  - ▶ `__LINE__` ligne courante dans le fichier source
  - ▶ `__FILE__` nom du fichier source
  - ▶ `__DATE__` date de compilation du programme
  - ▶ `__TIME__` heure de compilation du programme
  - ▶ `__STDC__` à 1 si implémentation conforme à ansi

```
#include <stdio.h>

int main (void) {
    printf("fichier %s", __FILE__);
    printf(" compilé le %s à %s\n", __DATE__, __TIME__);
    printf("Ligne %d\n", __LINE__);
    printf("Ligne %d\n", __LINE__);
    return 0;
}
```

\*.c

```
olivier@valrose:~ $ ./pp
fichier pp.c compilé le Mar 13 2025 à 09:31:35
Ligne 6
Ligne 7
```

SHELL

- ▶ Nous avons déjà vu :
  - ▶ `#ifdef NOM_MACRO`
  - ▶ `#else`
  - ▶ `#endif`
- ▶ Il existe aussi :
  - `#ifndef NOM_MACRO` pour voir si une macro n'est pas définie
  - `#if CONST` où `CONST` est un entier (booléen) connue à la compilation.
    - ▶ `CONST` peut être soit un entier, soit une macro valant un entier.
  - `#error MESSAGE` arrête la compilation et affiche le message.
  - `#warning MESSAGE` produit un warning avec le message

- 🍃 Partie I. Le préprocesseur
- 🍃 Partie II. Usage avancé préprocesseur
- 🍃 **Partie III. Fichier Headers**
- 🍃 Partie IV. Compilation séparée
- 🍃 Partie V. La bibliothèque standard (ANSI)
- 🍃 Partie VI. Autres bibliothèques
- 🍃 Partie VII. Table des matières



- ▶ Le code suivant ne compile pas :

```
int main(void) {  
    return est_paire(5);  
}  
  
/* retourne 1 (true) si n pair  
   retourne 0 (false) si n impair */  
int est_paire(int n) {  
    return 1-(n%2);  
}
```

\*.c

- ▶ Dans la fonction `main`, la fonction `est_paire` n'est pas encore définie
- ▶ Il suffit de déclarer la fonction avant la variable.

```
/* retourne 1 (true) si n pair et 0 (false) si n impair */  
int est_paire(int n); /* Déclaration */  
  
int main(void) {  
    return est_paire(5);  
}  
  
int est_paire(int n) { /* Définition */  
    return 1-(n%2);  
}
```

\*.c

- ▶ Est-ce vraiment utile ? Oui !
- ▶ Permet d'utiliser des fonctions définies dans un autre fichier.
- ▶ On pourrait déclarer les fonctions en début de fichier.
  - ▶ ce n'est évidemment pas pratique

```
#include <stddef.h> /* nécessaire pour size_t */  
  
void *malloc_prof (size_t taille);  
void *calloc_prof (size_t nb, size_t taille);  
void *realloc_prof (void *ptr, size_t taille);  
void free_prof (void *ptr);  
  
/* votre code ci-dessous */  
int main(void) {  
    int* t = malloc_prof(sizeof(int));  
    return 0;  
}
```

toto.c

- ▶ En pratique les déclarations externes sont mises dans un fichier *header*
  - ▶ on le nomme avec l'extension \*.h
  - ▶ on y déclare les fonctions que l'on peut utiliser.

```
#include <stddef.h> /* nécessaire pour size_t */  
void *malloc_prof (size_t taille);  
void *calloc_prof (size_t nb, size_t taille);  
void *realloc_prof (void *ptr, size_t taille);  
void free_prof (void *ptr);
```

stdprof.h

- ▶ Il suffit de l'inclure avec la directive du préprocesseur `#include`

```
#include "stdprof.h"  
  
int main(void) { ... }
```

toto.c

- ▶ Où est-ce que le compilateur cherche le fichier \*.h
  - ▶ si le nom est entre guillemets "lib.h" : dans le répertoire courant
  - ▶ si le nom est entre chevron <lib.h> : dans les répertoires système
  - ▶ sous GNU/Linux c'est le répertoire /usr/include/
  - ▶ On peut le préciser sous gcc avec l'option `-Ichemin`

- ▶ Des déclarations de fonctions
- ▶ Des déclarations de type.
  - ▶ Problème : un type ne peut-être définis qu'une fois!
  - ▶ Or un même fichier en-tête peut être inclus plusieurs fois.
  - ▶ Comment faire pour ne déclarer qu'une fois ?
- ▶ Le préprocesseur !
  - ▶ le code ne s'exécute que si la macro `MONHEADER` n'est pas définie.
  - ▶ la macro est alors définie
  - ▶ Cela garantie l'unicité de l'exécution du code.

```
#ifndef MONHEADER  
#define MONHEADER  
typedef ...  
#endif
```

\*.C

- ▶ On définit les types

```
#ifndef TYPESPROF_H
#define TYPESPROF_H

#define ABSENT -1
typedef enum { QCM, PARTIEL, EXAMEN } devoir;
typedef struct {
    char* nom;
    char* prenom;
    int qcm, partiel, examen;
} etudiant;
#endif
```

typesprof.h

- ▶ On déclare les fonctions qu'il faudra écrire.

```
#include "typesprof.h"
float moyenne(etudiant* bob);
void noter(etudiant* bob, devoir test, int note);
```

prof.h

```
#include "typesprof.h"
etudiant* nouvel_etudiant(char *nom, char *prenom);
void afficher(etudiant* etu);
```

etudiant.h

- ▶ La bibliothèque avec les fonctions pour le type étudiant :

```
#include <stdio.h>
#include <stdlib.h>
#include "typesprof.h"
#include "etudiant.h"

etudiant* nouvel_etudiant(char *nom, char *prenom) {
    /* À vous de l'écrire*/
}

void afficher(etudiant* etu) {
    /* À vous de l'écrire*/
}
```

etudiant.c

- ▶ La bibliothèque avec les fonctions pour l'enseignant :

```
#include <stdio.h>
#include "typesprof.h"
#include "prof.h"

float moyenne(etudiant* bob) {
    /* À vous de l'écrire*/
}

void noter(etudiant* bob, devoir test, int note) {
    /* À vous de l'écrire*/
}
```

prof.c

- ▶ Le programme final :
  - ▶ on utilise des fonctions définis dans `etudiant.c` et `prof.c`
  - ▶ et des types définis dans `typesprof.h`

```
main.c
#include <stdio.h>
#include <stdlib.h>
#include "typesprof.h"
#include "prof.h"
#include "etudiant.h"

int main(void) {
    etudiant* kevin = nouvel_etudiant("Glandouillou", "Kévin");
    noter(kevin,QCM,5);
    noter(kevin,PARTIEL,6);
    noter(kevin,EXAMEN,7);
    afficher(kevin);
    printf("--\nMoyenne : %.2f\n",moyenne(kevin));
    return 0;
}
```

- ▶ Et maintenant ? Comment on compile le tout ?

- 🍃 Partie I. Le préprocesseur
- 🍃 Partie II. Usage avancé préprocesseur
- 🍃 Partie III. Fichier Headers
- 🍃 **Partie IV. Compilation séparée**
- 🍃 Partie V. La bibliothèque standard (ANSI)
- 🍃 Partie VI. Autres bibliothèques
- 🍃 Partie VII. Table des matières



- ▶ On va d'abords compiler séparément chaque fichier \*.c
  - ▶ avec l'option de compilateur `-c`
  - ▶ les fichiers créés sont des fichiers objets (\*.o).
- ▶ Puis on va lier ces fichiers avec notre programme principale (avec le `main`)
  - ▶ on parle d'édition de lien
  - ▶ le programme utilisé par `gcc` s'appelle `ld`.
  - ▶ utile pour comprendre les messages d'erreur du compilateur!
- ▶ Si vous oubliez le `main`, c'est `ld` qui r le !

- ▶ On commence par ajouter toutes les lignes nécessaires.
  - ▶ Nous n'avons besoin que de trois appels à gcc

Makefile

```
all:  
gcc -Wall -ansi -pedantic -c prof.c  
gcc -Wall -ansi -pedantic -c etudiant.c  
gcc -Wall -ansi -pedantic etudiant.o prof.o main.c -o programme
```

- ▶ On découpe pour ne reconstruire que le nécessaire.
  - ▶ On indique pour chaque règle ses dépendances.

Makefile

```
all: programme

programme: main.c prof.o etudiant.o typesprof.h
    gcc -Wall -ansi -pedantic etudiant.o prof.o main.c -o programme

prof.o: prof.c prof.h typesprof.h
    gcc -Wall -ansi -pedantic -c prof.c

etudiant.o: etudiant.c etudiant.h typesprof.h
    gcc -Wall -ansi -pedantic -c etudiant.c
```

- ▶ Ajoutons des variables.
  - ▶ permet de modifier rapidement les options de compilations.

```
CC = gcc
OPTIONS = -Wall -ansi -pedantic

all: programme

programme: main.c prof.o etudiant.o typesprof.h
    $(CC) $(OPTIONS) etudiant.o prof.o main.c -o programme

prof.o: prof.c prof.h typesprof.h
    $(CC) $(OPTIONS) -c prof.c

etudiant.o: etudiant.c etudiant.h typesprof.h
    $(CC) $(OPTIONS) -c etudiant.c
```

Makefile

- ▶ Ajoutons une règle pour la compilation des bibliothèques.
  - ▶ `$<` représente le fichier source (ici le `%.c`)

```
CC = gcc
OPTIONS = -Wall -ansi -pedantic

all: programme

programme: main.c prof.o etudiant.o typesprof.h
    $(CC) $(OPTIONS) etudiant.o prof.o main.c -o programme

%.o: %.c %.h typesprof.h
    $(CC) $(OPTIONS) -c $<
```

Makefile

- ▶ Ajoutons une variable pour les bibliothèques
  - ▶ Ainsi l'ajout d'une nouvelle bibliothèque ne changera qu'une ligne

```
CC = gcc
OPTIONS = -Wall -ansi -pedantic
OBJETS = prof.o etudiant.o

all: programme

programme: main.c $(OBJETS) typesprof.h
    $(CC) $(OPTIONS) $(OBJETS) main.c -o programme

%.o: %.c %.h typesprof.h
    $(CC) $(OPTIONS) -c $<
```

Makefile

- ▶ On définit deux nouvelles variables
  - ▶ SOURCE : l'ensemble des \*.c du répertoire.
  - ▶ OBJETS : les même fichiers en remplaçant le .c par .o
  - ▶ Crée un fichier main.o.

```
CC = gcc
OPTIONS = -Wall -ansi -pedantic
SOURCE = $(wildcard *.c)
OBJETS = $(SOURCE:.c=.o)

all: programme

programme: $(OBJETS) typesprof.h
    $(CC) $(OPTIONS) $(OBJETS) -o programme

%.o: %.c %.h typesprof.h # ERREUR : main.h n'est pas définie
    $(CC) $(OPTIONS) -c $<
```

Makefile

- ▶ Comment gérer automatiquement les dépendances pour les fichiers \*.h ?
  - ▶ On ajoute une option à GCC pour qu'il calcule ces dépendances (-MMD)
  - ▶ On inclut les dépendances nouvellement trouvées. (-include \*.d)

```
CC = gcc
OPTIONS = -Wall -ansi -pedantic
SOURCE = $(wildcard *.c)
OBJETS = $(SOURCE:.c=.o)

all: programme

programme: $(OBJETS)
    $(CC) $(OPTIONS) $(OBJETS) -o programme

%.o: %.c
    $(CC) $(OPTIONS) -MMD -c $<

-include *.d
```

Makefile

```
olivier@valrose:~ $ cat prof.d
prof.o: prof.c typesprof.h prof.h
olivier@valrose:~ $ cat main.d
main.o: main.c typesprof.h prof.h etudiant.h
```

SHELL



- ▶ Notre makefile compile tous les fichiers \*.c du répertoire courant.
  - ▶ Il ne faut qu'une seule fonction **main**
  - ▶ Peut-être réutilisé dans d'autres projets
  - ▶ `make clean` supprime les fichiers inutiles

```
CC = gcc
OPTIONS = -Wall -ansi -pedantic
EXECUTABLE = programme
# Ne pas modifier ci-dessous
SOURCE = $(wildcard *.c)
OBJETS = $(SOURCE:.c=.o)
all: $(EXECUTABLE)

$(EXECUTABLE): $(OBJETS)
    $(CC) $(OPTIONS) $(OBJETS) -o $(EXECUTABLE)
%.o: %.c
    $(CC) $(OPTIONS) -MMD -c $<
clean:
    rm -vf *.o *.d
-include *.d
```

Makefile

```
olivier@valrose:~ $ make
gcc -Wall -ansi -pedantic -c -MMD etudiant.c
gcc -Wall -ansi -pedantic -c -MMD main.c
gcc -Wall -ansi -pedantic -c -MMD prof.c
gcc -Wall -ansi -pedantic etudiant.o main.o prof.o -o programme
```

SHELL

- 🍃 Partie I. Le préprocesseur
- 🍃 Partie II. Usage avancé préprocesseur
- 🍃 Partie III. Fichier Headers
- 🍃 Partie IV. Compilation séparée
- 🍃 **Partie V. La bibliothèque standard (ANSI)**
- 🍃 Partie VI. Autres bibliothèques
- 🍃 Partie VII. Table des matières

- ▶ En C ANSI, il existe une modeste bibliothèque standard.
  - `signal.h` : permet de gérer les interruptions (signaux)
    - ▶ Ctrl-C
    - ▶ Erreurs arithmétique (division par zéro)
    - ▶ Erreurs d'accès mémoire illicite
    - ▶ Permet d'envoyer de tel signaux.
    - ▶ Permet de lancer une fonction quand un signal est reçu.
  - `time.h` : pour la gestion du temps (date, heures, minutes, secondes)
  - `setjump.h` : permet de faire des goto mais en plus sale (saut en dehors de la fonction initiale : très fortement déconseillé).
  - `stdarg.h` pour définir une fonction avec un nombre variables d'arguments (comme `printf`). Peu utile et laborieux.

- ▶ Fonctions de classification (ne gère pas l'Unicode)
  - ▶ renvoie 0 si la réponse est négative et un autre entier sinon.

```
int islower(int c); /* c est-il une minuscule ? */
int isupper(int c); /* c est-il une majuscule ? */
int isdigit(int c); /* c est-il un chiffre décimale ? */
int isxdigit(int c); /* c est-il un chiffre hexadécimale ? */

int isalpha(int c); /* c vérifie-t-il islower ou isupper */
int isalphanum(int c); /* c vérifie-t-il isalpha ou isdigit */

int isprint(int c); /* c est-il imprimable (dont espace) ? */
int isgraph(int c); /* c est-il imprimable (sauf espace) ? */

int iscntrl(int c); /* c est-il un caractère de contrôle ? */
int ispunct(int c); /* c est-il un symbole de ponctuation ? */
int isspace(int c); /* c est-il un symbole d'espacement, retour
/* à la ligne, tabulation, etc. */
```

\*.C

- ▶ Fonctions de conversion

```
/* renvoie c converti en majuscule si C est minuscule, sinon c */
int toupper(int c);
/* renvoie c converti en minuscule si C est majuscule, sinon c */
int tolower(int c);
```

\*.C

- ▶ Possède de nombreuses fonctions (pour l'ASCII) : à savoir écrire par cœur!
- ▶ Petit panorama non exhaustif.

\*.c

```
/* copie n octets entre deux zones mémoire, qui ne doivent pas  
se superposer */  
void *memcpy(void *dest, const void *src, size_t n);  
  
/* remplit une zone mémoire de la répétition d'un caractère */  
void *memset(void *, int, size_t);  
  
/* concatène la chaîne src à la suite de dest */  
char *strcat(char *dest, const char *src);  
  
/* cherche un caractère dans une chaîne et renvoie un pointeur sur  
le caractère, en cherchant depuis le début */  
char * strchr(const char *, int);  
  
/* compare deux chaînes lexicalement */  
int strcmp(const char *, const char *);  
  
/* copie une chaîne de caractères d'une zone à une autre */  
char *strcpy(char *toHere, const char *fromHere);  
  
/* retourne la longueur d'une chaîne caractères */  
size_t strlen(const char *);
```

- ▶ Contient les fonctions `malloc`, `free` et `exit`
- ▶ Possède aussi deux fonctions pour générer un peu d'aléatoire :
  - ▶ Non sécurisée!

```
/* Retourne un nombre entier pseudo-aléatoire entre 0 et RAND_MAX  
   RAND_MAX vaut au minimum 32767 */  
int rand(void);  
  
/* Prend seed comme amorce de la nouvelle séquence de  
   nombres pseudo-aléatoire. L'amorce initiale vaut 1;*/  
void srand(unsigned int seed);
```

- ▶ Nombreuses fonctions mathématiques

```
sin(x)  
cos(x)  
pow(x,y)  
sqrt(x)
```

\*.c

- ▶ Pour compiler, il faut ajouter `-lm` au compilateur gcc.

- ▶ Vous la connaissez bien : `scanf`, `printf`



- ▶ Permet de savoir les tailles des types selon les machines.
  - limits.h Définit les valeurs max et min pour chaque type d'entier.

```
MAX_INT ; /* Plus grand int possible */  
CHAR_BIT /* Nombre de bit dans un char (normalement 8)*/
```

\*.c

- float.h Même principe, mais pour les décimaux.

```
FLOAT_MAX ; /* Plus grand float possible */  
DBL_EPSILON /* Plus petite différence entre deux doubles*/
```

\*.c

- ▶ Permet de tester une condition et d'échouer si elle n'est pas satisfaite
  - ▶ utile pour savoir si un indice ne dépasse pas la taille d'un tableau.
  - ▶ mais potentiellement coûteux à l'exécution.
- ▶ Définit la fonction `void assert(int expression);`

```
#include <assert.h>  
  
assert(i<10);
```

\*.c

- ▶ La fonction `assert` est ignorée si la macro `NDEBUG` est définie.  
`gcc ... -DNDEBUG ...`

- ▶ pour avoir une fonction **f** avec un nombre quelconque d'arguments.
- ▶ On déclare à l'intérieur de **f** une variable `ap` de type `va_list`.

```
/* f a deux argument obligatoire et potentiellement  
plusieurs facultatifs */  
type f(type0 arg0, type1 *arg1,...) {  
    va_list pa;
```

\*.c

- ▶ On initialise `pa` avec `va_start` (avec `arg1` dernier argument nommé)

```
/* arg1, pointeur vers le dernier argument nommé*/  
va_start(pa, arg1);
```

\*.c

- ▶ On récupère le suivant en indiquant son type avec `va_arg(pa, type)`
  - ▶ c'est au développeur de savoir quand s'arrêter.

```
entier = va_arg(pa, int); /* entier est de type int */
```

\*.c

- ▶ On termine par `va_end` dans la fonction **f**

```
va_end(pa);
```

\*.c

- 🍃 Partie I. Le préprocesseur
- 🍃 Partie II. Usage avancé préprocesseur
- 🍃 Partie III. Fichier Headers
- 🍃 Partie IV. Compilation séparée
- 🍃 Partie V. La bibliothèque standard (ANSI)
- 🍃 **Partie VI. Autres bibliothèques**
- 🍃 Partie VII. Table des matières

- ▶ Pour éviter le flou autour des types entiers : `stdint.h`
- ▶ Dans ce qui suit, vous pouvez remplacer 8 par n'importe quelle valeur N parmi : 8, 16, 32, 64
  - entier d'exactly 8 bits : `int8_t`
    - ▶ sur certaines architectures et pour certains N peut être non défini.
  - entier d'au moins 8 bits : `int_least8_t`
    - ▶ toujours défini : optimise la taille (plus petite taille valide).
  - entier d'au moins 8 bits : `int_fast8_t`
    - ▶ toujours défini : optimise la vitesse
    - ▶ il peut être parfois plus rapide de travailler sur 32 bits que 8 bits
- ▶ Existe en version non signée :
  - ▶ `uint8_t`, `uint_least8_t` et `uint_fast8_t`.

- ▶ `getopt.h` : pour créer une commande Unix avec paramètres optionnels :
  - ▶ `nl -x --color blue`
  - ▶ `nl --hexa` et `nl -x` sont équivalents
  - ▶ `nl -hx` et `nl -xh` sont équivalents à `nl --help --hexa`

```
int option_index = 0;

struct option long_options[] = {
    {"color",  required_argument, 0, 'c' },
    {"hexa",   no_argument,       0, 'x' },
    {"help",   no_argument,       0, 'h' }
};
```

\*.c

- ▶ On appelle :

```
c = getopt_long(argc, argv, "c:xh",
                long_options, &option_index);
```

\*.c

- ▶ Si l'option prend un argument, `getopt_long` le stocke dans `optarg`.
- ▶ Format : "option avec argument:option sans argument"

Merci pour votre attention

Questions



Hello world !



# Cours 6 — Modularité et compilation

Annonces

## Partie I. Le préprocesseur

Variable codée en dure

Les macros et le préprocesseur

Les macros avec arguments

Les macros ne sont pas des fonctions

Les macros ne sont toujours pas des fonctions

## Partie II. Usage avancé préprocesseur

Options du compilateur

Un exemple : debug

Conditionnelle de macro

Makefile

Macros prédéfinies

Autres directives

## Partie III. Fichier Headers

Déclarations et définitions de fonctions

Pourquoi déclarer des fonctions?

Fichiers header

Que contiennent les fichiers d'en-tête?

Exemple : les fichiers en-têtes

Exemple : les fichiers sources

Exemple : la fonction main

## Partie IV. Compilation séparée

Principe

Makefile : le début

Makefile : les dépendances

Makefile : les variables

Makefile : les règles automatiques

Makefile : les objets

Makefile : automatisation

Makefile : dépendances et fichiers en-tête

Makefile : perfection?

## Partie V. La bibliothèque standard (ANSI)

Pour les curieux

`ctype.h`

`string.h`

`stdlib.h`

`math.h`

`stdio.h`

`limits.h` et `float.h`

`assert.h`

`stdarg.h` (à lire chez vous)

## Partie VI. Autres bibliothèques

Types entiers explicites : `stdint.h`

Arguments en ligne de commande

## Partie VII. Table des matières